



**A METHODOLOGY FOR SIMULATING THE
JOINT STRIKE FIGHTER'S (JSF)
PROGNOSTICS AND HEALTH
MANAGEMENT SYSTEM**

THESIS

Michael E. Malley, Captain, USAF

AFIT/GOR/ENS/01M-11

DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

20010619 047

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the U. S. Government.

AFIT/GOR/ENS/01M-11

A METHODOLOGY FOR SIMULATING THE JOINT STRIKE FIGHTER'S (JSF)
PROGNOSTICS AND HEALTH MANAGEMENT SYSTEM

THESIS

Presented to the Faculty

Department of Operational Sciences

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the
Degree of Master of Science in Operations Research

Michael E. Malley, B.S.

Captain, USAF

March 2001

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

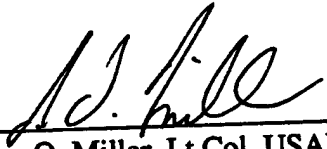
AFTT/GOR/ENS/01M-11

A METHODOLOGY FOR SIMULATING THE JOINT STRIKE FIGHTER'S (JSF)
PROGNOSTICS AND HEALTH MANAGEMENT SYSTEM

Michael E. Malley, B.S.

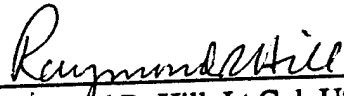
Captain, USAF

Approved:



John O. Miller, Lt Col, USAF (Chairman)

1 Mar 01
date



Raymond R. Hill, Lt Col, USAF (Member)

1 Mar 01
date

Acknowledgments

This research would have suffered without the help and guidance of several individuals. I would like to thank my advisor, Lt Col J.O. Miller for his patience and honesty throughout this process. I'm also grateful for the input provided on several occasions by Dr Ken Bauer and Lt Col Ray Hill. Besides helping with the content of the research, each of these three kept me focused on the end goal. I would also like to thank Gary Smith, Gerald Williams, and Squadron Leader Richard Friend who all provided input, and data, used in this research. My goal is that this research is true to all of your inputs.

I would like to thank my loving wife for supporting me throughout the thesis process. She graciously took over the housework and diaper changing duties while I worked on this project. Thank you for listening to my thesis problems, proofreading documents, and putting up with my frustration. You're a wonderful wife and mother.

Mike Malley

Table of Contents

Acknowledgments	iv
List of Figures	vii
List of Tables	viii
Abstract	ix
I. INTRODUCTION.....	1
Background	3
Problem Statement	6
Research Objectives	7
Scope and Assumptions	7
Methodology and Expected Results	9
Thesis Organization	10
II. LITERATURE REVIEW	11
Introduction.....	11
Maintenance Strategy	12
Aircraft Prognostics Management	13
<i>JSF PHM,</i>	13
<i>Boeing 777 and DS&S Maintenance Management.</i>	16
Engine Prognostics	17
<i>JetSCAN</i>	17
<i>JSF Engine Prognostics.</i>	18
<i>Digital Signal Processing.</i>	19
Simulation.....	22
<i>Predictive Maintenance Simulation.</i>	22
<i>ALSim.</i>	23
Artificial Neural Networks	24
Conclusions.....	26
III. METHODOLOGY	27
Introduction.....	27
Sensor Data	28
<i>Existing Datasets.</i>	28
<i>Signal Generator.</i>	30
<i>Signal Generator Algorithm.</i>	33
Signal Processing.....	36
Signal Generator Verification and Validation	41
Neural Network Construction.....	42
Building Failure and False Alarm Distributions.....	44

ALSim Modifications	46
Conclusions.....	48
IV. ANALYSIS AND RESULTS	49
Introduction.....	49
Signal Generator Output	49
Neural Network Analysis.....	53
Early Failure Detection Time versus False Alarm Tradeoff.....	57
<i>Batch Size Study</i>	57
<i>Neural Network Assignment Rule</i>	61
Failure Detection Time and False Alarm Rate Results.....	64
<i>Failure Detection Time</i>	64
<i>False Alarm Rate</i>	67
ALSim Inputs.....	68
<i>Failure Detection Time</i>	68
<i>ALSim Operation</i>	72
Conclusions.....	73
V. CONCLUSIONS AND RECOMMENDATIONS.....	75
Further Study	77
Appendix A. Signal Generator Code	79
Appendix B. ALSim PHM Code	98
Bibliography	104
Vita.....	106

List of Figures

Figure 1. JSF ALS Components	6
Figure 2. Capt Rebulanan's PHM Implementation	9
Figure 3. Neural Network Construction.....	25
Figure 4. Signal Generator User Interface	31
Figure 6. Wear-in Signal.....	50
Figure 7. Flight Signal	50
Figure 8. Failure Signal.....	51
Figure 9. Complete Signal	52
Figure 10. Batched Signal.....	53
Figure 11. Trained Neural Network Confusion Matrices	55
Figure 12. Batch Size of 10 vs Batch Size of 20	58
Figure 13. Batch Size 10 versus Batch Size 5	58
Figure 14. ROC Curve Example.....	62
Figure 15. Affect of Changing Classification Rule	63
Figure 16. Failure Time Distribution for 1500 Life.....	66
Figure 17. Failure Time Distribution for 3700 Life.....	66
Figure 18. Failure Detection Time as Percent of Life (1).....	70
Figure 19. Failure Detection Time as Percent of Life (2).....	71

List of Tables

Table 1. Signal Generator Component Life Settings	37
Table 2. Signal Generator Simulation Settings.....	38
Table 3. Signal Generator Output Form	40
Table 4. Component Lifetimes	53
Table 5. Batch Size 10 and 20 False Alarm Rates.....	59
Table 6. Batch Size Affect on Failure Detection Time.....	60
Table 7. Failure Time Distribution	65
Table 8. Scaling Lifetime.....	69
Table 9. Failure Detection Time Trend.....	69
Table 10. False Alarm Scaling.....	72

Abstract

The Autonomic Logistics System Simulation (ALSim) was developed to provide decision makers a tool to make informed decisions regarding the Joint Strike Fighter's (JSF) Autonomic Logistics System (ALS). The benefit to ALS is that it provides real-time maintenance information to ground maintenance crews, supply depots, and air planners to efficiently manage the availability of JSF aircraft. This thesis effort focuses on developing a methodology to model the Prognostics and Health Management (PHM) component of ALS. The PHM component of JSF is what actually monitors the aircraft status.

To develop a PHM methodology to use in ALSim a neural network approach is used. Notional JSF prognostic signals were generated using an interactive Java application, which were then used to build and train a neural network. The neural network is trained to predict when a component is healthy and/or failing. The results of the neural network analysis are meaningful failure detection times and false alarm rates. The analysis presents a batching approach to train the neural network, and looks at the sensitivity of the results to batch size and the neural network classification rule used. The final element of the research is implementing the PHM methodology in ALSim.

A METHODOLOGY FOR SIMULATING THE JOINT STRIKE FIGHTER'S (JSF) PROGNOSTICS AND HEALTH MANAGEMENT SYSTEM (PHM)

I. Introduction

The Joint Strike Fighter (JSF) is the next generation fighter aircraft being developed to meet air threats in the year 2010 and beyond. The JSF incorporates the latest technology in the aircraft industry. One of the revolutionary developments is the Autonomic Logistics System (ALS), which is designed to efficiently manage the JSF maintenance and logistics programs. The ALS is possible because of two concepts. First, the JSF uses well-placed sensors and diagnostics to detect impending faults and uses reasoning algorithms to determine their causes; and second, this information quickly disseminates throughout the logistics infrastructure (Scheuren, 1998: 2). The Prognostics and Health Management (PHM) system on board the aircraft performs the first function, and the Joint Distributed Information System (JDIS), a strategic communication system, performs the latter.

The advantage of the ALS approach is that it reduces the time a JSF aircraft spends in non-mission capable status. The PHM system minimizes the time maintenance personnel need to diagnose a failure or fault in the weapon system, and JDIS reduces the time it takes to requisition replacement parts. The JSF ALS begins with the continuous monitoring of the aircraft's systems to detect and diagnose deteriorating performance or system failure. Sensors and control algorithms used in the PHM isolate the faults and communicate maintenance requirements to personnel on the flight line and in the logistics infrastructure using JDIS. Time is saved because maintenance crews perform only limited diagnostic tests when the aircraft returns to base, and the JDIS system sends immediate notice of aircraft status to the logistics infrastructure.

This technology will substantially alter Department of Defense (DoD) logistics policy, which is reactive and/or preventative in nature. Current fighter weapon systems cannot be fault detected until the mission is completed and the aircraft returns to base. Maintenance personnel then enter a detailed diagnostic series to isolate the problem. The logistics organizations cannot proactively begin locating a replacement part until after the maintenance crew successfully identifies the fault. The time required to complete these tasks just adds to the length of time the aircraft will be unavailable to perform the mission.

Captain Rene Rebulanan (GOR-00M) built a computer simulation of the ALS and its components called the Autonomic Logistics Simulation or ALSim. ALSim was used to show that the JSF ALS would lead to significantly higher weapon system availability versus current strike aircraft (Rebulanan, 2000: 55-56). ALSim has all the necessary elements to model the JSF ALS, the next step is to add a level of fidelity to some of the

components to make the simulation more useful for decision making. This thesis effort will focus on developing a methodology to model the PHM system and then implement it in ALSim.

Background

The Joint Strike Fighter is the next generation strike aircraft being developed jointly by the US Air Force, Navy, Marine Corps as well as several foreign allies. The goal is to develop an aircraft that can be used by each military service, including US allies, without compromising the performance of the aircraft by tailoring it towards a specific service. To meet affordability goals, each aircraft will be built using approximately 70% - 90% common parts, with minor exceptions to meet service specific requirements (JSF Program office, 2000: slide 7). For example, the Navy model JSF has more robust landing gear and vertical take-off capability to accommodate the aircraft carrier environment. Currently the program is in the early design phase, in which two contractors are building prototype JSF aircraft for flight test. The prototype aircraft are flight demonstrators only and do not include the ALS capability that will exist on the operational aircraft. One contractor will be selected based from this fly-off to proceed to the Engineering and Manufacturing Development acquisition phase. The first combat ready JSF will be delivered in fiscal year 2007 (Hough, 1999: slide 4).

One revolutionary aspect of the JSF is the development of the ALS. Autonomic logistics is the state-of-the-art in logistics management and is largely driven by the increasing complexity of DoD weapon systems. Older aircraft, such as the

McDonnell Douglas F-15 and General Dynamics F-16, were designed to isolate component failures using built-in-tests. The diagnostic cycle currently follows a path that starts with fault detection, moves to fault isolation, and finally failure analysis and fault treatment. This requires aircraft-specific support equipment to isolate failures, which is costly, takes a great deal of time, and is terribly inaccurate. "Can Not Duplicate" and "Tests-OK" situations now account for 50% of DoD repair results (Blemel, 1998: 1). A Can Not Duplicate condition exists when the maintenance crew cannot repeat the detected fault or error on the ground. Similarly, a Test-OK is when a piece of equipment fails a diagnostic test, but when tested again the equipment shows no sign of fault. Furthermore, using the F-15 as an example, approximately 25 maintenance hours are required for each flight hour of operation (Blemel, 1998: 1-2). With shrinking military maintenance budgets this scenario obviously is no longer viable. The ALS system is designed to reduce the amount of time the JSF spends in non-mission-ready status, reduce the cost of maintaining the weapon system, and provide a robust diagnostic suite that minimizes detection inaccuracies.

The autonomic logistics system, specifically the PHM, will provide continuous monitoring of the operation, health, and safety of the JSF. If during flight the PHM detects degraded performance from any component or subsystem, it will isolate the fault, predict the failure time, and schedule the necessary maintenance tasks to be completed. Once again, maintenance actions will be forwarded to the logistics chain before the aircraft returns to base. This continuous diagnostic coverage and fault detection capability is being designed to reduce, and hopefully eliminate, the time required by the

maintenance crew to diagnose the problem once the aircraft lands. This added time is used to locate spare parts, or refresh the maintenance crew on the repair/replace task.

One of the enabling technologies used in PHM is prognostics. Where diagnostics are used for fault detection and isolation, the goal of prognostics is to predict failure. The JSF recently completed a series of seeded fault engine tests to examine the prognostics of the aircraft engine. To perform the test an engine was fitted with several sensors being proposed for use on the JSF aircraft, including the engine distress monitoring system sensor, ingested debris monitoring system sensor, the wear site sensor, and the oil-line sensor. Fault conditions were induced on the engine and the sensor detection capability was measured. The control logic for real-time diagnosis of the fault is not yet completed, but it essentially works in the following manner. Operational limits are placed on the sensor's reading, and out of limit conditions signify degraded performance and impending failure (Powrie and Fisher, 1999: 12).

Similar prognostic measures are used on the Boeing 777 aircraft. The 777 Central Maintenance Computer (CMC) uses an, "abductive control algorithm to encode a cause and effect relationship between faults and their symptoms and then interpret fault scenarios against these relationships" (Felke, 1994: 1). The 777 model defines fault scenarios, their symptoms, and repair actions for each line replaceable unit (LRU) on the aircraft. The algorithms within the CMC use this hard coded information to interpret conditions on the aircraft and identify appropriate repair actions for the faults it detects (Felke, 1994: 1).

This thesis effort builds on a thesis done last year by Capt Rene Rebulanan (2000). Capt Rebulanan built a first-order model (ALSim) that defines all the

components of the ALS and gives them appropriate functionality (Figure 1). The model consists of depot supply, base supply, JDIS, PHM, LRU, flightline maintenance, and flightline supply classes built using the Java programming language. Using the model, Capt Rebulanan was able to simulate the JSF ALS and conclude that the ALS leads to higher aircraft availability (Rebulanan, 2000: 55-56).

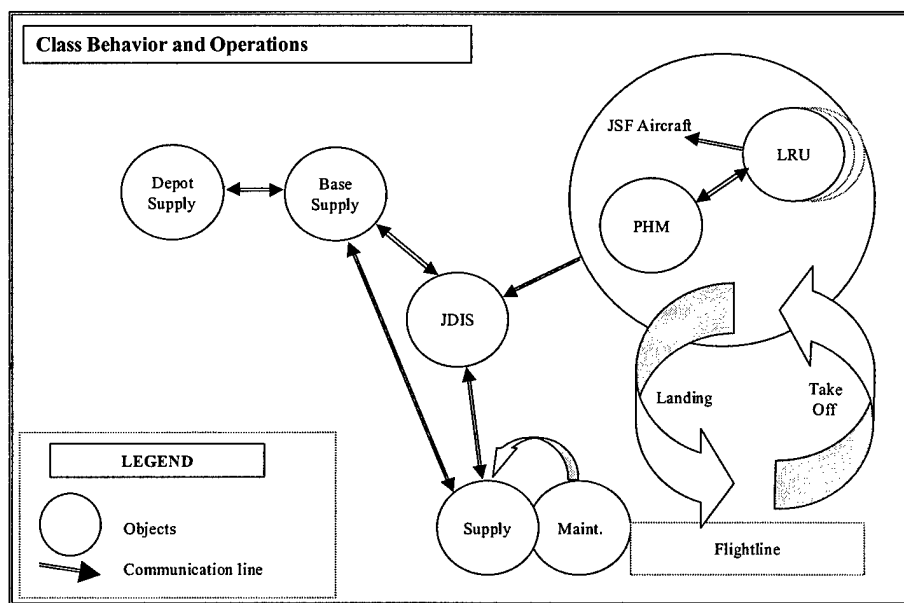


Figure 1. JSF ALS Components

Problem Statement

The ALS strategy is new to DoD and as such little is known about its actual capabilities or the demands it places on the existing logistics infrastructure. ALSim provides an initial framework of the JSF autonomic logistics system that can be used to analyze the ALS components and characterizes the operational system. The next step is

to add a level of fidelity to the model. ALSim's components really only provide the framework for the model and need further definition. The component of the model with the least understanding is the PHM, and a methodology for modeling the PHM component of the system needs to be developed and implemented to fully understand the capabilities of ALS.

Research Objectives

The goal of this research is to build on a previously developed simulation model that can be used to predict JSF support requirements and weapon system availability. Previous research focused on building an initial, high-level simulation model of the ALS. The next step is to develop a methodology for modeling the PHM component of the ALS. This enhanced PHM methodology will be added to ALSim. The original model performance measures will be used to track the ALS ability to meet JSF program goals of achieving a 25% increase in combat sortie generation rate relative to current strike aircraft. The ALSim measure of performance linked to sortie generation is JSF availability.

Scope and Assumptions

The typical mission profile used in this simulation will be for the aircraft to take off for a mission, perform the mission, and then return to base. During the mission the PHM will monitor the condition of the JSF aircraft and monitor aircraft systems for

degraded performance. Any maintenance actions generated will be passed through the logistics chain using the JDIS element of the simulation. The appropriate maintenance actions will be taken once the aircraft returns to base, either immediately or, if parts are unavailable, at the earliest time the parts become available.

The model of the system will simulate the operations of one JSF Wing and corresponding support organizations needed to support one Wing. Also, the JSF aircraft is too complex to simulate every part. This thesis will focus on key line replaceable units (LRU) associated with the engine subsystem.

The focus of this thesis effort will be on developing and implementing accurate PHM components of the JSF autonomic logistic system. The logistics chain consisting of the JDIS, base supply shop, depot supply, and flightline maintenance will not be altered dramatically. The model currently implements base supply, depot supply, and flightline maintenance elements according to Air Force or DoD policy. The only changes to this aspect of the simulation will be in the interface with the PHM components.

Since the JSF aircraft is still in development, system reliability data cannot be used. Where possible, the simulation will use state-of-the-art reliability and logistics information. When this is not possible existing Air Force aircraft data will be used. To the maximum extent possible the simulation will allow for easy entry of JSF data, as it becomes available.

As the model is developed additional assumptions will need to be made to simplify the effort. These will be listed and discussed where appropriate.

Methodology and Expected Results

This thesis effort will focus on developing a methodology for building the PHM component of the JSF ALS, and then implementing that strategy in ALSim. Figure 2 shows the ALSim PHM implementation. For a given LRU, a random draw determines when the LRU will fail (have no remaining life). Based on the settings in ALSim the JSF aircraft is programmed to detect this failure some percent of time before the failure occurs (Rebulanan, 2000: 29). For example, if the failure time is 100 and ALSim is set to detect a failure at 95% of the component life, then the PHM component of the JSF will detect the failure at time 95. This approach assumes perfect, 100% failure identification.

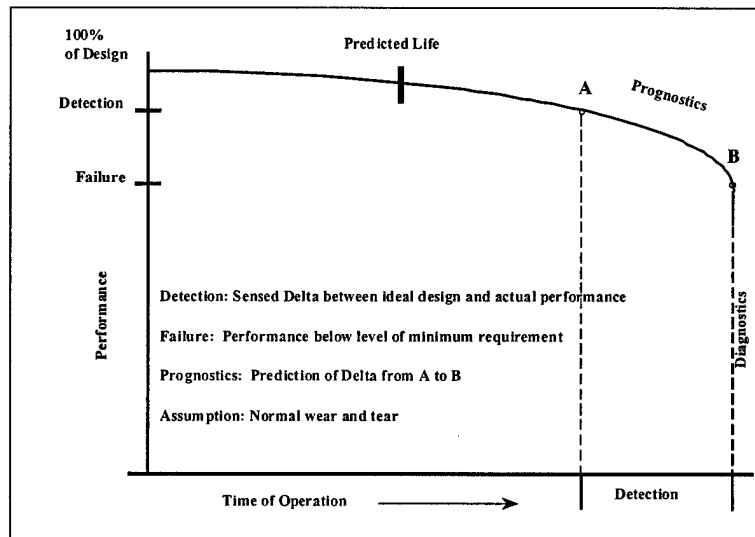


Figure 2. Capt Rebulanan's PHM Implementation

This thesis effort uses multivariate analysis techniques to build a distribution of predicted failure detection times as well as adding the capability to model false alarms (predicting failure when the system is healthy). A PHM Signal Generator, built using Java, is used to generate the data to be analyzed. The final step is to integrate the PHM methodology in ALSim to better characterize the system.

Thesis Organization

This thesis document is organized into five chapters. The second chapter is a detailed literature review of the pertinent subjects to this thesis. First, aircraft prognostics of the JSF and other systems are discussed. Second, several ongoing data-analysis techniques for prognostics data are presented. Third, existing prognostic simulation models are presented, and finally, a short description of neural networks is included.

Chapter three describes the PHM methodology developed for this thesis. It includes a detailed description of the PHM Signal Generator built in Java. A technique is developed for inputting the PHM signal into an artificial neural network that is trained to predict when a component is failing. This chapter also includes an explanation of how the failure time distribution is built. Finally, a discussion of how the methodology is implemented in ALSim is provided.

The fourth chapter includes the results from the PHM Signal Generator, the neural network prediction, and the ALSim modifications. The results are explained and interpreted. The final chapter includes study conclusions.

II. Literature Review

Introduction

The purpose of this literature review is to search for examples of how aircraft maintenance systems similar to PHM have been built and modeled. To accomplish this task there are several groups of topics that need to be researched for use in developing a PHM methodology and then implementing in ALSim. The first topic is maintenance strategy. The JSF PHM clearly pushes the state-of-the-art in maintenance management, and understanding how that maintenance strategy is used is important for building the system. Second, aircraft prognostics/diagnostic maintenance systems need to be researched. The JSF system is still in development; so existing systems with ALS-like capability need to be analyzed as examples of how ALS can be implemented. Third, since the engine prognostics package is the most developed and is the focus of this thesis, pertinent information on engine diagnostic technology needs to be presented. This includes PHM implementation techniques and possible methods to analyze the engine data collected from the Air Force Research Laboratory. A discussion on neural networks is also included that provides a basic background of neural network construction and includes research on the use of neural networks with prognostics. Finally, this thesis uses the Java programming language to build a simulation model. Research on simulation

models built using the Java programming language may provide ideas and methods for implementing a new PHM methodology.

Maintenance Strategy

Air Force maintenance strategies have evolved over the past 50 years. Walls identifies the three maintenance strategies used today as reactive maintenance, preventative maintenance, and predictive maintenance (1999: 151-153). Reactive maintenance is a passive strategy where maintenance actions are not taken until the system fails (Walls and others, 1999: 152). This is obviously a very easy strategy to implement, however, it results in unpredictable system performance and system availability. Furthermore, since system performance is unpredictable the maintaining organization must have a large maintenance force and deep inventory of spare parts. Preventative maintenance removes some of the unpredictable nature of system failures, by scheduling maintenance actions at predetermined intervals (Walls and others, 1999: 152). For example, car manufacturers recommend changing the oil filter and oil in a vehicle every 3000 miles. An organization can plan and forecast based on a preventative maintenance strategy. One downfall of this approach is that the maintenance is performed on a system that is still in good working order. Time spent working on a system that hasn't failed is really unproductive, furthermore, if a part is replaced based simply on schedule it may have useful life left. This has led rise to the predictive maintenance strategy. Predictive maintenance relies on the principle that, "99% of all machine failures are preceded by the certain signs, conditions, or indications that a failure

was going to occur (Knapp, 1996: 1)". Predictive maintenance tries to isolate what these signs or conditions are and use them to dictate maintenance actions (Walls and others, 1999: 153). This strategy increases the probability that a component will remain in service for most of its useful life, and requires a minimal maintenance staff and spare part inventory. Obviously this approach is the most complex of the three strategies and requires complex understanding of the system and its failure modes. While each strategy is appropriate in certain situations, the Air Force is trying to implement a predictive maintenance strategy on the JSF program to reduce maintenance requirements.

Aircraft Prognostics Management

JSF PHM. The Joint Strike Fighter Prognostics and Health Management (PHM) system is the on-aircraft hardware and software that enable predictive maintenance. The PHM hardware consists of well-placed diagnostics and prognostics throughout the aircraft. To the maximum extent possible diagnostics that are already used on the aircraft will be used as part of the maintenance diagnostic suite. Scheuren notes,

Research with intelligent diagnostic systems has shown that small changes in the relationships or levels of the various variables (e.g. vibrations modes, temperatures, pressures, electrical resistance, etc.) that define the machine of interest are precursors to the failure that can be reliably used to predict future failure. (1998: 3)

If it is not possible to use diagnostics already intended to go on the aircraft, then maintenance specific diagnostics will be added to the design. The PHM software will implement artificial intelligence algorithms to isolate faults and predict failure time.

One approach for implementing PHM reasoning is the Evolvable Tri-Reasoner Integrated Vehicle Health Management System (Atlas and others, 1999: 1). This approach actually focuses on three separate reasoners for each aircraft subsystem, and one independent reasoner at the aircraft system level. The first subsystem reasoner is a diagnostic reasoner that is used to isolate faults and failures. The diagnostic reasoner records inputs from various diagnostics placed throughout the subsystem to determine the cause of a fault. This reasoner is trained the same way diagnostic reasoners in existing aircraft are built. Detailed failure modes and effects analysis is completed on each subsystem and programmed into the reasoner. The second reasoner is the prognostic reasoner. The prognostic reasoner relies on input from prognostics located throughout the subsystem to predict the useful life remaining in components of the subsystem. The idea behind a prognostic reasoner is that a component has a nominal component life curve which includes variability for each component. Based on where a component is on its "life curve" the prognostic reasoner can identify how much longer the component will continue to function (Atlas and others, 1999: 2-4). The final subsystem reasoner is an anomaly reasoner. The anomaly reasoner is used to classify off-nominal behavior. The anomaly reasoner collects off-nominal data that can later be used to update the prognostic or diagnostic reasoner (Atlas and others, 1999: 11).

The system level reasoner is the Reasoner Integration Manager (RIM). The RIM relies on input from the prognostic reasoner, diagnostic reasoner, and anomaly reasoner to determine if maintenance action is required. The RIM uses prognostic input to characterize where a component is on its life curve and how far away it is from its nominal curve. The diagnostic reasoner is used to isolate what is causing the failure and

the necessary repair action. Finally, if a component or subsystem is operating in an off-nominal condition the anomaly reasoner provides input about what could possibly be occurring in the system. The RIM uses input from all the reasoners to best characterize the state of a subsystem and recommend any necessary maintenance actions (Atlas and others, 1999: 9-10). The reasoning algorithms used on the JSF are intelligent mathematical models such as rule based reasoning, model based reasoning, cased based reasoning, neural networks, fuzzy logic, and genetic algorithms (Scheuren, 1998: 3).

The PHM algorithms are developed in a very methodical process. First, the failure modes and effects analysis of the aircraft systems and subsystems is performed. Obviously, the JSF is still in the development phase, so this analysis is solely based on engineering design. Second, the failure modes and effects analysis is used to determine which subsystems need to have health management capability. Finally, for those systems or components that merit health management capability, the failure modes and effects analysis is molded into an aircraft model. For example, if one engine failure mode is that a compressor blade becomes dislodged, the effects of that failure are linked to the cause and used in a reasoning algorithm (Scheuren, 1998: 3).

The JSF is still in the development phase and as such the reasoning algorithms are developed off of engineering design. The advantage to using neural networks, fuzzy logic, or other reasoning algorithms is that as the system matures, the PHM system improves (Scheuren, 1998: 4). As health data is accumulated on the aircraft, the reasoners will be retrained to report more accurate and helpful health information. This is truly the power of the PHM subsystem. If during flight testing for example, the test crew finds that a certain engine seal has a long wear-in time, the PHM can be trained to

recognize that anomalous behavior and not trigger a maintenance action. The PHM has the potential to define the operational envelope of the aircraft.

Boeing 777 and DS&S Maintenance Management. There are other examples of maintenance management systems being implemented in aircraft, such as the Boeing 777. The Boeing 777 system was developed in a similar manner as the JSF PHM. The abductive algorithms were written in parallel with the engineering design of the aircraft (Felke, 1994: 3-4). Boeing had to deal with several significant issues in developing the Central Maintenance Computer (CMC) in order to make it effective. First, the appropriate level of resolution needed to be determined. A model that is too general does not provide sufficient accuracy, and a model that is too detailed is difficult to build and maintain. A careful balance between these two extremes was maintained by balancing the fault detection requirement with the implementation and maintenance cost. When the cost was too high, alternative arrangements were sought (Felke, 1994: 4-5). The second issue that had to be dealt with was the natural time lapse between the start of component failure and notification to the CMC of this time. To overcome this time shift the model logic and algorithms were trained to include this phenomenon (Felke, 1994: 1-2). The final issue that had to be resolved was how to build the system while requirements were still being generated. In response Honeywell built a proprietary tool called the Data Capture Tool (DCT) (Felke, 1994: 3). The DCT shielded the airborne software from the details of the exceptional data items. Engineers responsible for each system used the DCT to enter their system data, which was then integrated into an aircraft level model. This process allowed the 777 and CMC design to occur in parallel with minimal rework required.

Another system that closely resembles the JSF autonomic logistics system in its entirety is Data Systems & Solutions Company (DS&S) real-time engine condition monitoring (ECM) system. The first airline to contract with DS&S to use the ECM system is German-based Condor whose fleet consists of 13 Boeing 757-300 aircraft, powered by Rolls-Royce RB211-535 engines (DS&S, 26 July 2000 press release). During flight, engine data is transmitted from the aircraft to a DS&S engine health center in the United Kingdom for processing. In July 2000, this data went online via the Internet site enginedatacenter.com so the DS&S, Condor, and Rolls Royce have real-time data availability (DS&S, 26 July 2000 press release). The online system tracks engine performance data, recommends maintenance actions based on flight data, and even tracks spare part shipments.

Engine Prognostics

JetSCAN. JetSCAN system is currently being used by the British Royal Air Force to examine oil samples for material chemical composition, size, and shape (morphology). The system was initially designed to solve a Tornado RB 199 engine problem where metal chip detectors were failing to detect significant bearing material losses. The JetSCAN system uses a scanning electron microscope to analyze a collected oil sample. Using material knowledge of the system being sampled, the oil analysis helps maintainers determine location, type, and rate of wear that is occurring in the engine. This information is used to calculate failure risk, develop trends, and estimate the time to failure. JetSCAN has increased the magnetic chip detector removal interval by 100%

(went from 25 to 50 flight hours), which reduces costs by eliminating unnecessary engine removals and preventing on-wing engine failures (*www.ds-s.com*, 2000: n. pag.).

The JetSCAN system is also seeing trial use on US Air Force installations. The Air Force hopes that JetSCAN can solve the #1 safety issue in Air Force Material Command; the F-16, F-18 F-100 engine #4 bearing problem (Pomfret, 2000: 3). The JetSCAN technology is a good example of predictive maintenance using information on the aircraft (in this case engine oil) to predict aircraft failure. The drawback, of course, is that oil samples still must be taken and analyzed off-board.

JSF Engine Prognostics. In addition to normal engine diagnostics, one JSF contractor has developed PHM specific prognostics. Pratt-Whitney performed the first of two-planned Seeded Fault Engine Tests (SFET) in 1999 to test three engine health-monitoring systems. The first system is the Engine Distress Monitoring System (EDMS) that examines the engine exhaust gas. The system detects the electrostatic charge associated with debris present in the exhaust gas. The system provides real-time warning of incipient fault conditions. Similarly, the Ingested Debris Monitoring System (IDMS) detects electrostatic charge of debris in the inlet to the engine. The final system is the oil monitoring system that detects electrostatic charge of debris in the engine oil system (Powrie and Fisher, 1999: 11-12).

The three systems all work using the same principle; under normal operating conditions a healthy sensor detects some level of electrostatic charge. If the electrostatic signal has significant deviation from this normal value then somewhere in the system component level degradation is occurring. The JSF PHM engine algorithms account for

normal increase in electrostatic charge that occurs over time as the system is 'worn-in' (Powrie and Fisher, 1999: 12-13).

The SFET test yielded significant results. The sabotaged engine test runs successfully demonstrated the capabilities of the above-mentioned sensors. Powrie notes that some faults went undetected and there were false detections (1999: 18). For example in the ingestion tests, the IDMS was tested to discriminate between Category 0, 1, and 2 debris. Category 0 includes non-damaging debris. During each test, 10-90 category 0 detections were observed, primarily due to insect ingestion. Category 2 refers to damaging debris. There were cases where category 0 debris triggered category 2 detection, but the exact number of instances is not included in the article. Finally, category 1 refers to debris whose threshold has not been defined in the control logic database or reasoning. Basically, if a category 1 condition exists, the control algorithms can't discern what entered the engine. As the debris database fills out, these occurrences should become more rare. One other significant finding of the oil system tests was that it is hard to diagnose the system if more than one component is failing. The system detects a fault, but cannot reason what caused the detection (Powrie and Fisher, 1999: 18-19).

Digital Signal Processing. Signal processing background is important to this thesis because actual F-100 engine data has been obtained which can be used to help build a simulation methodology for the PHM component of the JSF ALS. As part of an Air Force sponsored Small Business Innovative Research (SBIR) contract in 1999, Frontier Technology analyzed actual aircraft engine data using interesting techniques. The purpose of Frontier's work is to determine whether existing engine instrumentation can effectively be used to detect degraded engine performance and predict engine

performance. The focus of their work was on using existing engine instrumentation because it saves the time and money of retrofitting AF aircraft with additional hardware or software. This approach obviously fits within the realm of predictive maintenance, because Frontier is trying to predict engine failure using various engine diagnostics (Keller and Eslinger, 1999: 7-10).

The data that Frontier collected came from 17 test runs of an F-100-PW-220 engine performed at the Arnold Engineering and Development Center (AEDC). The data include 77 parameters instrumented on an AF F-100 engine. The test data was not specifically collected to investigate engine prognostics; rather it was collected as part of an experiment to characterize the F-100 engine in different flight conditions. During the final test run the engine failed due to a portion of a sixth stage high-pressure compressor blade detaching. Because the test data was collected at many different flight conditions, Frontier decided to start by analyzing transient data in the tests because it represented the only repeatable data in the experiment. Specifically, 9 transient test runs where the engine went from idle to military power in 2 minutes were analyzed (Keller and Eslinger, 1999: 7-10).

To analyze the data Frontier developed two methods to test independent parameters and other metrics made up of several parameters. The first technique is all parameter visualization. It looks at relative changes of a parameter over time using color (Keller and Eslinger, 1999: 10). If a parameter changes color very little during a test or a series of tests, a blue or green bar represents the parameter. If, however, the parameter begins to change values significantly the color changes to red. The idea behind this approach is that if the system fails then somewhere in the system something must be

happening. A parameter that doesn't change will not be useful in predicting a particular failure. The second method that Frontier developed is called all parameter trending. This technique provides a single number characterization of deviations in measured parameters based on transient data from a test run relative to an established basis group of test runs (Keller and Eslinger, 1999: 10-12).

One of the metrics used by Frontier is hyperspace mean deviation. For the technique the nine test runs were divided into 3 groups of 3 runs where the first group contained tests 1, 2, and 3; the second group contained tests 4, 5, 6; and the third group contained test runs 7, 8, and 9. For a given parameter or group of parameters the average value of the nine test runs is calculated and then the average for each of the subgroups is calculated. Frontier speculated that the third subgroup should have a larger distance from the overall mean, which could be used to show incipient failure (Keller and Eslinger, 1999: 12-13).

To this point Frontier's results are promising. The all parameter visualization technique works well at seeing how individual parameters change throughout a test. The hyperspace mean deviation technique initially looked promising, however, the company noticed that the results of the analysis were very correlated to parameters controlled by the test crew that ran the tests. The correlation makes it difficult to determine if the experimental settings caused the metric to change, or if the metric is in fact indicating an impending failure. The Air Force Research Laboratory has extended the Frontier contract, so the company plans to further examine non-correlated metrics it has developed (Keller and Eslinger, 1999: 27-29).

Simulation

Predictive Maintenance Simulation. Szczerbicki has developed a predictive maintenance simulation for an industrial condition-monitoring service group that performs inspection, vibration, oil, and wear debris analysis (1998: 482). The model simulates the monitoring service and is built to optimize the manning scale and instrument resources for the service. The model is built using the SLAMSYSTEM simulation language and defines a unique methodology for modeling predictive maintenance (Szczerbicki and White, 1998: 481).

The model is built as a traditional discrete event simulation. The model creates entities at specified intervals that represent the vibration analysis and oil analysis for the system. Resources such as maintenance staff, analysis machinery, and facility management act upon the entities. The model includes logic for routing each analysis task through the simulation. For example, the probability of a vibration analysis being created that is corrupted is governed by a Triangle(0, .08, 1.0) distribution; if the entity is corrupted the model logic knows to immediately generate another vibration analysis entity (Szczerbicki and White, 1998: 492). All of this logic is driven by probability distributions; no form of artificial intelligence is implemented.

The maintenance simulation ran for 12 simulated weeks and was used to determine staffing levels for the maintenance activity. Basically, the model was used for parametric analysis by changing the number and skill level of crew assigned to the different maintenance groups. Using this approach, the model was able to successfully optimize the maintenance crew configuration. The model was verified and validated

using simple entity flow tests and a bottom-up dynamic testing strategy. Individual components of the model were tested before the model was put together in whole. Once the whole model was built Szczervicki used the TRACE animation option to verify the model (Szczerbicki and White, 1998: 492-497).

ALSim. Air Force Captain Rene Rebulanan (GOR-00M) built a top-level JSF ALS simulation model. He modeled the system using the Silk collection of Java simulation classes. His model includes a JSF class, a scheduler class, a PHM class, a supply class, a base supply class, a depot supply class, a JDIS class, and a maintenance class. These classes basically mirror the elements of the JSF maintenance and supply chain. Each class includes the methods necessary to simulate operation of the JSF ALS in its entirety. ALSim's current PHM class simply calculates the time when the PHM detects degraded performance of a JSF line replaceable unit as a constant percentage of the items mean time between failure (Rebulanan, 2000: 29). ALSim is designed to be a flexible simulation, and items such as the PHM detection time (percentage listed above) can easily be changed. Captain Rebulanan's thesis varied the detection time between 90%, 95%, and 99% of the components life. As an example, for an LRU life of 1000 hours, ALSim was run using PHM detection times of 900 hours, 950 hours, and 990 hours.

ALSim's measures of performance are JSF availability, the cumulative number of sorties taken per 24-hours for the entire simulation period, and the accumulated wait-time due to supply. The first performance measure is the proportion of time a JSF aircraft is available for mission. Each JSF object created includes an availability time dependent statistic that tracks this value. The second parameter is the cumulative number of JSF sorties taken per 24-hours for the simulation period. This is collected using a simple

count observational statistic. Finally, the accumulated wait-time due to supply is determined by tracking the time a JSF aircraft object spends in the queue waiting for a part (Rebulanan, 2000: 42-43).

Captain Rebulanan's thesis simulated six months (183 days) of JSF ALS operation using four ALS enabled aircraft and four non-ALS enabled aircraft. The simulation was replicated 30 times and statistical analysis run comparing the four ALS aircraft to the four non-ALS aircraft. The results showed that the ALS enabled aircraft were statistically different then the non-ALS enabled aircraft. For all three measures of performance the ALS aircraft showed better performance then the non-ALS aircraft: aircraft availability is higher, cumulative sorties flown is higher, and wait-time due to supply is lower (Rebulanan, 2000: 42-53).

Artificial Neural Networks

One of the most common methods for detecting aircraft failures appears to be the application of neural nets. In the aircraft failure arena the neural network acts as a classification algorithm – classifying a healthy system or a failing system. Figure 3 shows the physical representation of a neural network. A neural network is a collection of nodes that process signals. Each node takes a weighted sum of its input to establish its net input and then transforms the input using a linear, sigmoid, or hyperbolic function (Bauer, 2000: 4). A linear combination of the weights applied to the inputs yields the output. The goal of training a neural network is to optimize the weights such that the error between the actual output and the predicted output is minimized.

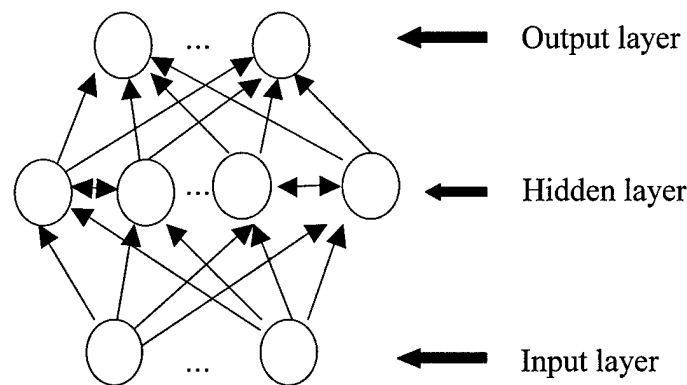


Figure 3. Neural Network Construction

To train a neural network the data must be split into three pieces: training, training-test, and validation. The validation data is not used until the network is fully trained. The training and training-test data are used to actually train the neural network. During one epoch all the training data is passed through the network and weights are calculated that minimize the assignment error (actual to predicted output). After each epoch the training-test data is passed through the resulting network to determine if the network has been sufficiently trained. If the prediction error is too high in the training-test dataset, then another epoch of training data is fed into the network to further train it. When the training-test dataset error is minimized, the network is sufficiently trained.

There are two big advantages to training a neural network as a predictor. First, because nonlinear sigmoid and hyperbolic transformations are used the classification accuracy of neural networks is generally higher than linear predictors. Second, unlike discriminant analysis or basic regression, neural networks do not require independent data. This removes the hassle of checking for independence and then transforming the

data to comply with the assumption. These advantages make neural networks very useful and simple to use.

One example of neural networks being applied in maintenance classification is an F-16 Fire Control Radar (FCR) study done in 1996. The neural network design included three layers (input, hidden, and output) and was used to classify the faulty avionics system. The goal of the study was to be able to classify with 90% accuracy whether a radar system was a “lemon”, “bad actor”, or “normal”. All three conditions indicate a faulty system or failing system. A system is classified as a lemon if it is classified as a healthy system in different aircraft. A bad actor identifies a system that is classified as healthy on an aircraft during one diagnostic test and subsequently classified as failing on another diagnostic test. Normal identifies an always-faulty system. The study concluded by achieving a maximum classification accuracy of 80%.

Conclusions

The JSF Autonomic Logistics System is revolutionary and pushes the state-of-the-art in logistics and maintenance management. Although the system is still being designed, enough literature about the ALS and comparable systems exist to make considerable headway towards better defining PHM in ALSim. Furthermore, subscale test data on the JSF engine prognostics will be very useful. Several data analysis techniques were presented that can be used to analyze failure data and develop the PHM methodology.

III. Methodology

Introduction

The following methodology will be used to more accurately reflect PHM operation in ALSim. This is not to say that the current PHM component is incorrect or that it doesn't provide sufficient information about the JSF ALS. In fact, ALSim was used to provide meaningful results about ALS capability as part of Capt Rebulanan's thesis. This thesis effort focuses on making the PHM component more realistic. The results will hopefully be able to help the Air Force further understand the capability of ALS, and develop a meaningful Concept of Operations (ConOps) for the JSF aircraft.

This chapter focuses on the method developed and implemented to more accurately model the JSF PHM component of ALS. The approach to solving the problem breaks down into three processes. The first of these is to gather data on the JSF PHM system that can be used to build a model of the system. Actual system data is the best to analyze the system and build a representative model, but for reasons stated in the next section this is not possible at this time. The second process is to analyze the data that was gathered to determine how it could best be used to model PHM. To add realism to the PHM component a neural network will be used to analyze the PHM data and characterize the system's performance. From the neural network, a database is built that maintains

probability of false alarm, probability of detection, and the time associated with false alarm or detection for specific component failure times. The final process is to input the database back into ALSim to make it more accurate. The following sections break down these three processes into further detail to clearly show the methodology used to model the PHM.

Sensor Data

Existing Datasets. The best way to accurately model the PHM component of ALS is to use real-world PHM data. Unfortunately for the JSF system this is difficult because the aircraft is still largely an engineering drawing. Although flight-testing of the Boeing and Lockheed Martin prototypes is underway, these prototypes have no PHM capability for cost reasons. The best alternative to using JSF data is to use data from a similar system. As the literature discussed in Chapter 2 outlines, the greatest abundance of PHM-type data is in aircraft engines. Two alternate data sources were found that potentially could be used to model PHM. Ultimately neither of these two sources of data was used for reasons discussed below.

The first source of data mentioned in Chapter 2 is F-100-PW-220 engine data collected several years ago at Arnold Engineering and Development Center (AEDC). The engine was not being tested for fault detection reasons and as such did not have any suite of fault diagnostics. The tests were simply performance tests run to characterize the engine. On what became the final test, the engine experienced a compressor blade

detachment that prematurely ended the test series. The tests were run at different operating conditions, although across the 17 tests there were nine “idle to military power” test sequences. The last “idle to military power” test sequence was run during test 14. To analyze this data all 77 engine parameters were differenced between the nine different sequences to see if any parameter may have been able to predict the failure. Unfortunately, even within a specific test sequence the parameters varied too much to make any meaningful conclusions. Furthermore, any difference that was seen could not be decoupled from possible test run conditions (operating altitude for example). This data set did not prove to be useful.

The second set of data analyzed was the JSF seeded fault engine test (SFET1 and SFET2). This data looked very promising because the tests were specifically run to test PHM components at the development level. The data came from the Aeronautical System Center Propulsion Directorate, which supports the JSF program office. Once the data was in hand it became clear that the PHM components were not included, and that the only data available was typical engine diagnostics. Furthermore, the test plan from the test series was not written so there was no formal documentation of any given test. The data was accompanied only by a several page spreadsheet that had one-line goals for each test run. Using the spreadsheet, several test configurations were found that differed only in the aspect that the first test was run without the fault present and the second run with the fault present. Again the engine parameters were differenced to look for trends. Two problems developed. First, there was no log of when a fault was induced into the system and consequently no way to know if a parameter was indeed detecting a fault in the system. Second, the tests were all run at different operating conditions (throttle position

for example), which made it impossible to tell if a parameter changed due to operating condition or incipient failure.

To overcome these deficiencies a simulation was developed to generate a PHM component sensor signal that could be analyzed. The drawback to this approach is that the simulation had to be very generic since it could not be baselined with PHM data. With this in mind a Java program was written to generate a sensor signal for a given set of input conditions. The resulting Signal Generator uses input from a graphical user interface (GUI) to generate a PHM sensor signal.

Signal Generator. The Signal Generator is a very generic simulation that builds a sensor signal based on user input. The only “data” available to build a representative simulation were several journal articles that described in some detail how the PHM system will notionally operate (Scheuren, 1998; Powrie, 1999). The key components drawn from the literature are the factors that could adversely affect the PHM’s ability to detect a failure. Using this information, the simulation builds a notional sensor signal.

The basis for the Signal Generator really comes from the ingested debris monitoring system (IDMS) and/or the engine distress monitoring system (EDMS). These sensors were discussed in Chapter 2, and are used as prognostics in the JSF engine to predict mechanical failures. Both sensors operate by measuring the electrostatic discharge of the gas that flows through the engine. Under normal operating conditions each sensor produces a baseline signal that represents the “normal” amount of electrostatic charge that exists in an engine. As an engine component degrades and sheds material the amount of electrostatic charge in the gas flow increases. The failing component can be isolated because different metals lead to different sensor readings.

Using the above information the baseline signal from the Signal Generator could be programmed. Obviously the process is time-dependent so a time-series process is used to build the sensor signal. The governing equation to build the signal is:

$$s_t = \mu + (c \times s_{t-1}) + \delta \quad (1)$$

where

- s_t is the signal value at time t
- μ is the mean of the signal
- c is a coefficient that induces correlation between signal readings (set at .8)
- s_{t-1} is the signal value at time $t - 1$
- δ is a standard normal noise term $\sim \text{Normal}(0,1)$

To further describe how the simulation works it is necessary to look at the GUI interface (Figure 4).

JSF PHM Signal Generator

The JSF PHM Signal Generator builds a PHM sensor signal based on the selected settings.

Signal Nominal Mean: 100

Component Life: 300

Component MTBF: 3000

Run Simulation

Number of Replications: 30

Signal Sensitivity to Component Wearin

0 3 15 0 15 50

Wearin time as % of life Wearin occurrence rate (%)

Signal Sensitivity to Changing Flight Conditions

0 10 50

Adverse flight condition occurrence rate (%)

PHM Failure Prediction

0 20 20

Variability in failure start

Figure 4. Signal Generator User Interface

The user interface includes four text boxes that can accept user input and four slider values that can be manipulated. The first text box is for the signal nominal mean. This can be any value, but the simulation currently is programmed for 100 – if the value is changed other values in the code need to be changed accordingly. Using a mean of 100 the steady state signal is centered on 500. The second box allows the user to enter the component life. This is the time when the component for which the signal is being generated has completely failed (JSF is grounded). The simulation is programmed to randomly determine this value, but for the purposes of this thesis it was easier if the component life could be entered. Rather than wait for a random draw to yield a life that could be used in data analysis, a meaningful value can be entered. The third text box is the exponentially distributed mean of the component's failure time – commonly called mean time between failures (MTBF). Most of the prognostics being developed for the JSF will operate at 10s to 100s Hertz, however for the Signal Generator as built, seconds or minutes make sense as the time units. The final text box is for the user to input the number of replications to be run at a specific setting.

The slider bars are essentially used to alter the mean (μ in Equation 1) for different flight conditions. The PHM literature revealed three primary concerns for a signal change. The first is component wear-in. An engine seal, for example, takes time to set, which will be reflected in a prognostic sensor signal. The first slider value under “Signal Sensitivity to Component Wear-in” is used to determine how long the component is sensitive to wear-in conditions. The slider represents a percent of the component MTBF entered by the user. The second slider is used to determine how sensitive the component is to wear-in anomalies. This value again is a percent. As an example if the

slider bar value is 30, then 30 percent of the resulting wear-in portion of the signal will be off-nominal.

The slider under “Signal Sensitivity to Changing Flight Conditions” changes μ in Equation 1 due to changes in flight conditions. Consider a JSF aircraft going from idle throttle to full afterburner. The mechanical stress this places on the engine temporarily leads to higher electrostatic readings. This slider value, similar to the “Signal Sensitivity to Component Wear-in” slider is represented as a percent. The final slider is the “Variability in Failure Start” slider, and is used to indicate the variability in the failure onset time. In building the Signal Generator a point in time has to be chosen to start the failure portion of the sensor signal. This slider allows the user to input the variability of when that failure portion of the signal should begin. As discussed below the Signal Generator is programmed to start the failure portion of the signal at 90% of its life. This slider value controls the variability of that start time – it represents the standard deviation of the start time and ranges from 0 to 2 (0 to 0.02 standard deviations).

Signal Generator Algorithm. The Signal Generator is built using two Java classes. The first is **JSFGui** and holds the user interface and the main program. The second class is **SensorSignal** which has five methods used to build the sensor signal and process it. The program is setup as a Java application that can be run on any operating platform.

The first part of the signal generated is the wear-in portion. The wear-in portion of the sensor is calculated by multiplying the “Wearin time as % of life” slider value (percent) by the component MTBF. The wear-in portion of the sensor is thus independent of the actual life of the component – for a given component it’s the same no

matter the components failure time. The mean of Equation 1 is changed based on the values of the “Wearin occurrence rate” slider and the “Adverse flight condition” slider. The sum of these slider values is compared to a uniform random draw (between zero and one). The sum of both slider values is used because at this stage of the component’s life it is exposed to both wear-in conditions and flight conditions. If the random draw is less than the slider values, then Equation 1 is modified using the following equation:

$$\tilde{\mu} = \mu + (\rho \times 2.25) \quad (2)$$

where

$\tilde{\mu}$	is the adjusted mean
μ	is the user entered mean
ρ	is a Uniform(0,1) random draw

An element of randomness is included because the signal change will be different based on the event. The value of 2.25 is used because it keeps the steady state signal from exceeding 510, which is used in the program to indicate component failure. When a random draw is performed and it triggers a “wear-in event”, the mean is adjusted using Equation 2, then four iterations of the signal are calculated using Equation 1 (with the adjusted mean). Four time units are chosen because the events are supposed to be transient in nature. This can easily be changed if necessary. As an example of the wear-in process consider the following: if the two slider values are set at 15 (.15) and 10 (.10), the random draw is compared to .25. If the random draw is .20, then the mean is adjusted and four signal measurements generated using the adjusted mean. The process of generating the wear-in portion of the signal is accomplished using the *wearinGenerator* method in the **SensorSignal** class.

The second portion of the signal that is generated is the “flight” portion. The flight portion of the signal represents that portion of the signal that is not influenced by wear-in conditions or failure conditions. To generate this portion of the signal the same algorithm is used as described above for the wear-in portion of the signal with one exception. The random number draw is compared to only the “Adverse flight condition” slider. If the random number is less than the slider percentage, Equation 2 is used to calculate the adjusted mean and then Equation 1 used to generate four signal measurements. There is nothing in the literature that suggests four measurements are correct or constant, however the goal is to simply model a transient event. The value could easily be changed if necessary. The flight portion of the signal is generated using the *flightGenerator* method in the **SensorSignal** class.

The last portion of the signal is the “failure” portion. Failure is defined when the signal measurement reaches 510 (assuming the starting mean is 100). To determine when the failure will start Equation 3 is used:

$$t_f = [(\phi \times f_s) + .9] \times l_c \quad (3)$$

where

t_f	is the length of time the signal is considered in failure state
ϕ	is a standard normal (0,1) random draw
f_s	failure slider divided by 1000 (range of 0 to .02)
l_c	is the component life

Equation 3 is used to determine when the failure portion of the signal will begin. The failure begin time is normally distributed with a mean of 90% of the component life and the standard deviation is user defined using the failure slider. Using this beginning

failure time, the slope of the failure signal is calculated using Equation 4. Finally Equation 5 is used to actually generate the signal.

$$\beta = \frac{510 - 500}{t_f} \quad (4)$$

$$s_t = s_{t-1} + [\beta \times (.7 + (.6 \times \theta))] \quad (5)$$

where

β	is the slope of the line between 510 and 500
θ	is a Uniform(0,1) random draw
s_t, s_{t-1}	same as defined in Equation 1

The uniform random draw in Equation 5 permits some randomness in the failure portion of the signal. For each failure signal measurement, the signal increment is uniformly distributed between 0.7 and 1.3 times the baseline signal increment.

The Signal Generator builds a notional sensor signal. Its purpose in this research is simply to provide data from which a PHM modeling methodology can be developed. When actual JSF sensor data is obtained the Signal Generator can be modified to accurately reflect the real world data.

Signal Processing

Once the signal is generated it needs to be analyzed to determine if and when component failure can be predicted. The goal, again, is to be able to predict impending failure. Discriminate analysis, and building an artificial neural network are the best potential analysis methods for analyzing the signal data. The basic approach of both

analysis methods is discriminating between various populations. The component in a healthy state and the component in a failure state are the two populations that need to be differentiated. Neural networks tend to have higher accuracy rates than discriminate analysis because neural networks use nonlinear functions. However, predicting healthy versus failing components is currently a one-dimensional problem (sensor signal). With this in mind the neural network analysis and discriminate analysis should lead to similar classification accuracies. As the JSF begins flight testing and operational use, extensive aircraft data will be gathered that can be used to predict aircraft failure. With a large collection of data a neural network has the best ability to classify the component as healthy or failing, so a neural network methodology will be developed that can be implemented later.

To successfully train the neural network it needs to be exposed to the full range of data. To achieve that goal the Signal Generator was run for 30 replications with a mean time between failure (MTBF) of 3000 at the various component lives listed in Table 1.

Table 1. Signal Generator Component Life Settings

Component Life	Cumulative Dist
300	0.10
700	0.21
1100	0.31
1500	0.39
2100	0.50
2700	0.59
3700	0.71
4900	0.80
7200	0.91
14100	0.99

The first column represents the component life (when the component fails), and the second column is the cumulative exponential distribution using the respective component life and mean MTBF. The following example should clarify how to interpret the data in Table 1. Using the exponential cumulative distribution with mean 3000, the probability that the component life will be less than 300 is ~0.10 or 10%. The table shows that the component lifetimes were chosen in 10% increments over the full range of possible lifetimes. This table is important for two reasons. First, it shows that the range of operating conditions for a component with MTBF equal to 3000 will be adequately covered. Second, the cumulative distribution information is useful in analyzing the data and for possibly extending the results to components of any MTBF.

The simulation settings for each component life are listed in Table 2.

Table 2. Signal Generator Simulation Settings

Parameter	Setting
Signal Mean	100
MTBF	3000
Replications	30
Wear-in time as percent of life	3 (0.03 or 3%)
Wear-in occurrence rate	15 (0.15 or 15%)
Adverse flight condition occurrence rate	10 (0.10 or 10%)
Variability in failure start	20 (0.02 or 2%)

Based on these settings the component will be subject to wear-in events for 90 time units, and the wear-in events will occur 25% of the time. The flight condition events will occur 10% of the time. Finally, the failure portion of the sensor signal is normally distributed about 90% of the component's life with standard deviation of .02%. For the way the Signal Generator is written these values were found to be reasonable starting positions.

The final element of processing the signal is putting it in a meaningful form to train the neural network. Feeding the raw data to the network is not realistic and furthermore not practical. The raw data is very transient in nature and the purpose of the PHM system is to diagnose the long-term health of the JSF aircraft. To achieve this effect the raw signal is averaged, or batched, to smooth out the signal. Notionally the batched mean of a healthy component will be less than the batched mean of a failing component. One experiment performed as part of this research is determining what batch size is most appropriate for accurately modeling PHM. Traditionally batching is used in simulation analysis when only one or two simulation runs can be performed. Batching the simulation run allows meaningful statistical analysis to be completed on the simulation. When choosing a batch size in a simulation environment the goal is generally to use a batch size large enough to get independent data (one batch is independent from another). For this research the batch size is chosen to balance two goals – minimal false alarms, and timely failure detection. The neural network will determine at what value a component can “safely” be classified as operating healthy in contrast to what value the component can be classified as failing. A method in **SensorSignal** was added to perform the batch operation (*batchSignal*).

The other element of signal processing involves identifying a batch as “healthy” or “failing”. The Signal Generator builds a three-column matrix that can be used in the neural network. The first column is the batched signal values. The remaining two columns are indicator variables for the health of the component. The first state of health is “healthy” (hereafter referred to as healthy) and takes on the value 1 if the component is healthy and 0 if the component is failing. The second state of health is “failing” (hereafter referred to as failing) and takes on the value of 1 if the component is failing and 0 if the component is healthy. Table 3 shows how a typical output file looks.

Table 3. Signal Generator Output Form

Batched Signal	Healthy	Failing
502.576	1	0
...
509.678	0	1

The Signal Generator algorithm distinguishes between healthy and failing based on the condition of the component at the end of a batch. If the component is healthy at the conclusion of the batch then the batch is labeled healthy. Similarly if the component is failing at the end of a batch then the batch is labeled failing. Using this definition, a batch of 10 that begins with 9 healthy signal measurements and finishes with one failing signal will be labeled as failing. By definition the Signal Generator classifies the wear-in and flight portion of the raw signal as healthy. The failure portion of the raw signal is

identified as failing. Consequently, any batch that includes the failure portion of the raw signal is labeled failing.

Signal Generator Verification and Validation

Before the Signal Generator was used it was verified and validated. Verification insures the program does what it is intended to do, and validation addresses whether the Signal Generator is representative of the real life system. To properly verify the system operates as intended three aspects of the system had to be checked. First, the process of generating the raw signal had to be examined to insure the generated signal properly used the user input. Second, the batching process needed to be verified. Finally, the process of appending the healthy and failing indicator status to the batched signal array had to be verified.

To insure the raw signal properly represented user input two approaches were used. All of the Java programming was done using Symantec's VisualCafe programming environment. VisualCafe has a very robust debugging capability that allows the user to run the Java application in a step-by-step fashion. Breakpoints can be added anywhere in the code and the debugger runs until it hits a breakpoint. At each breakpoint the user can look at the value of all the program variables. Using this capability, breakpoints were added to all the logic conditions and loops of the Signal Generator and the system was run multiple times to capture any errors. Several errors that would have otherwise gone unnoticed were corrected using this approach. The other verification step used was

simply graphing the generated signal to insure it looked as intended. Generating signals at extreme values and comparing the results graphically showed the algorithm works as intended.

Verification of the batching process and the process by which the indicator variables were added was much easier. Again, VisualCafe's internal debugging capability was utilized and proved helpful. Perhaps more convincing, however, the raw signal for a given run could be brought into Excel and manually batched and compared to the same batched signal generated by the Signal Generator. Similarly, the healthy and failing status could be added in Excel to the batched signal and compared to the Signal Generator output. If the Excel manual results differed from the Signal Generator then the algorithm had to be looked at again.

Validation of the Signal Generator is very difficult, and probably not meaningful. No data exists on the JSF PHM system against which to compare the Signal Generator output. Furthermore, the real purpose of the Signal Generator is to generate a notional signal that incorporates transient effects. It doesn't really matter what the signal looks like, so long as it incorporates the transient effects and has a definite failure status. Graphically examining both the raw and batched signal provides the best indication that the system produces a proper signal.

Neural Network Construction

To build the neural network, Statistical Neural Network Analysis Package (SNNAP) was used. SNNAP is a Microsoft Windows based product, which is very easy

to use. The mathematical analysis performed in building the neural network is transparent to the user, who merely selects criteria for constructing and training the network. The goal is to build a network using all the data generated using the Signal Generator. The resulting neural network can then be used to build a distribution of failure detection times and build a distribution of false alarm times.

To construct the neural network one input and two outputs were used. The input is simply the batched signal. The two outputs are indicator variables signifying if the sensor is measuring a healthy or failing component. For example if a batched signal represents a healthy component then the “Nominal Signal” output has a value of 1 and the “Failing Signal” output has a value of 0. The network has 10 nodes in the middle or hidden layer. The input node is transformed using a linear function, while the hidden layer and output layer nodes are transformed using a sigmoid function. The sigmoid function enables the neural network to build a relationship using a nonlinear equation.

One concern in using a neural network for this problem is that substantially more data are collected in the healthy state than in the failure state. This could potentially fool the neural network into classifying all the data as healthy or nominal. To prevent this from happening only a subset of the healthy data is used to train the network. The subset is generated by identifying the number of failure state data collected and then randomly selecting an equivalent number of healthy state data. For example if 100 failure data exist in the training set then close to 100 healthy data will be randomly selected to complement the failure data. The modified data set is broken into three pieces. The first piece represents the validation data that is never used to train the network; it is only used for validation and is 25% of the data. The remaining 75% is broken up into training data

and training-test data. Of the 75%, on a given training epoch, two-thirds of the data is used for training the network and one-third is used to test the network.

The built neural network uses a back propagation training method, which is not very efficient but will always converge to an optimum solution. A back propagation network trains by minimizing root mean square error (RMSE), where RMSE is the error between a predicted output value and the actual output value. Once the RMSE is minimized a confusion matrix is built of the training, training-test, and validation data. The confusion matrix measures the classification accuracy of the network. If the network is built with representative data then the confusion matrices for each data set should lead to similar classification accuracies.

Building Failure and False Alarm Distributions

Once the neural network is built the next step is to use it to build failure and false alarm distributions for a given component life. The first step to accomplish this is to run the Signal Generator at each component life for many replications and run the data through the trained neural network. For each batched mean the neural network will predict whether the component is healthy or failing. The neural network output is used to build the distribution of predicted failure time and the false alarm rate. To build a distribution of failure detection times a database is populated with the time when the neural network first detects incipient failure for a given component life. Across 30 replications for each component life setting, sufficient information is gathered to build a histogram of failure times. Using *Arena's Input Analyzer* the empirical data can be fit to

a normal distribution and the parameters calculated. A false alarm is defined as a batch that the neural network predicts is failing, but is actually healthy. The false alarm rate can then be determined by looking at the proportion of replications that have a false alarm. To build a false alarm distribution the times of false alarm for a given component life are cataloged and fit to a distribution. Figure 5 shows how the false alarm and failure detection time data are collected. The component life in the figure is 1500, and a section

A	B	C	D	E	F	G	H	I	J	K
Batch Size	10									
Life	1500									
Rep	1									
Rep	Time	False Alarm	Failure Detectic	Batched Sign	Healthy	Failing	pred_Healt	pred_Failin	pred_Healt	pred_Failin
2	1180	FA		504.778	1	0	0.064674	0.935898	0	1
2	1190			500.873	1	0	0.97275	0.027279	1	0
2	1200			498.872	1	0	0.999947	5.34E-05	1	0
2	1210			499.55	1	0	0.999817	0.000183	1	0
2	1220			501.399	1	0	0.95127	0.048769	1	0
2	1230			502.507	1	0	0.91581	0.084136	1	0
2	1240			500.373	1	0	0.993642	0.006354	1	0
2	1250			501.812	1	0	0.940322	0.059688	1	0
2	1260			502.628	1	0	0.908098	0.091836	1	0
2	1270			501.448	1	0	0.949803	0.050233	1	0
2	1280			500.827	1	0	0.975052	0.024973	1	0
2	1290			499.58	1	0	0.999799	0.000201	1	0
2	1300			500.323	1	0	0.994825	0.00517	1	0
2	1310			501.618	1	0	0.945174	0.054852	1	0
2	1320			497.403	1	0	0.999962	3.82E-05	1	0
2	1330			499.202	1	0	0.999921	7.98E-05	1	0
2	1340			501.108	1	0	0.961865	0.038178	1	0
2	1350			500.651	1	0	0.983536	0.016474	1	0
2	1360			500.516	0	1	0.989122	0.010879	1	0
2	1370			500.877	0	1	0.972596	0.027434	1	0
2	1380			501.556	0	1	0.946792	0.053238	1	0
2	1390			502.243	0	1	0.927862	0.072109	1	0
2	1400			502.875	0	1	0.884988	0.114924	1	0
2	1410			503.537	0	1	0.706455	0.293529	1	0
2	1420	FD		504.201	0	1	0.267679	0.733051	0	1
2	1430			504.832	0	1	0.056903	0.943633	0	1

Figure 5. Neural Network Output

of replication 2 is shown. Column C shows that at time 1180 the batched signal value is 504.778, which the neural network classifies as failing (FA). Column F and G, however,

show the actual batch status as healthy. Looking at this component life, there is now one false alarm out of 30 replications. The false alarm time is collected to build the false alarm time distribution. The second thing to notice in Figure 5 is when the neural network predicts the system is failing. At time 1420 the system detects the component is failing (FD), even though the failure portion of the signal actually began at time 1360. The failure detection time is annotated and when added to the failure detection times from the remaining replications will be sufficient to build a failure detection time distribution. The analysis results in three pieces of information that will be used in ALSim: false alarm rate, failure detection time, and false alarm detection time for a given component life.

ALSim Modifications

The goal of this thesis effort is to develop a more accurate PHM methodology to use in ALSim. The previous sections discuss the methodology employed to generate more realistic failure detection times and add the reality of false alarms. The result of this methodology is essentially a database of failure detection time distributions, false alarm probability, and false alarm occurrence time distributions. The next step is to add this information into ALSim and the logic to handle false alarm occurrences.

The ALSim PHM component currently works in a very simplistic manner. A random failure time is drawn and the JSF detects this failure at 95% of the drawn failure time. When the PHM component detects the failure (95% of random draw) it knows

when the actual failure time is and allows the aircraft to fly up until that failure time. Meanwhile the logistics chain is taking the necessary actions to repair the JSF aircraft. The ALSim PHM component has no false alarm allowance.

The existing PHM random failure draw will still be used. The resulting failure time will be compared to the database of failure times created. For values not explicitly included in the database, the failure detection time, false alarm probability, and false alarm occurrence time will have to be interpolated. Two additional random variate draws will have to be added to ALSim. The first will be to determine if the component will have a false alarm or if the component will operate until failure. This is simply implemented by comparing the result of a uniform (0,1) random draw to the false alarm probability for the failure time. If the random draw is less than the false alarm probability then the component will experience a false alarm. Otherwise the component will operate until the failure detection time. The second random draw will be to pull a false alarm occurrence time or a failure detection time. The distributions for these two events will be cataloged in ALSim.

The final modification to ALSim is adding the necessary logic for responding to false alarm occurrences. Nothing needs to be added for the failure detection, the existing logic can still be used. The only change is that the failure detection time will be variable rather than a fixed 95% of the LRU life. The concept of operations for the JSF logistics system is still largely undefined, particularly to the level of detail needed to handle false alarms. Obviously if a JSF aircraft detects failure the logistics chain begins. The way failures are handled in ALSim currently is that the PHM communicates with the JDIS which then starts the logistics process. There is no person in the loop – the repair process

is simply begun. Since the concept of operations is so immature, the false alarm logic will be implemented as if the false alarm were an actual failure. The repair process will be initiated and a healthy LRU exchanged for a new LRU.

Conclusions

The preceding sections detail a well-planned and appropriate methodology for modeling PHM. The data being used to implement the methodology is, by necessity, notional. Nonetheless this research focuses on the approach, and once actual data becomes available it will be straightforward to implement the proposed methodology. The Signal Generator is a tool that is designed with sufficient flexibility to provide meaningful data about PHM operation and its sensitivities. To a large extent this capability will not be exploited in this research. Additional research could and should be done using the Signal Generator to further understand PHM operation. The specific outcomes of this modeling approach will be more accurate failure detection times and realistic detection probabilities. Used in ALSim this information will be useful for developing the JSF Concept of Operations (ConOps), and for meaningfully characterizing the JSF ALS capability.

IV. Analysis and Results

Introduction

This chapter includes the results of this research effort. First, results from the Signal Generator are presented to illustrate how the Signal Generator works. Second, the neural network analysis is presented. Next, a trade study of the link between failure detection time and false alarm rate is presented. The trade study is necessary to characterize the relationship between failure detection time and false alarm rate. Next, the failure detection time data and false alarm rate data is presented and explained. The final failure and false alarm results are chosen for incorporation in ALSim. Finally, the ALSim implementation is presented.

Signal Generator Output

For purpose of illustration the Signal Generator code was modified for the initial run to break up the signal into its component pieces and show how the signal is put together. For this illustration the component life is 1500 and the remaining user settings are the same as defined in Table 2. Running the Signal Generator created a signal with the following characteristics: the wear-in portion of the signal includes the first 90 time units; the flight portion of the signal makes up time 91 – 1385; and the failure portion of the signal starts at 1386. Figure 6 shows the wear-in portion of the signal.

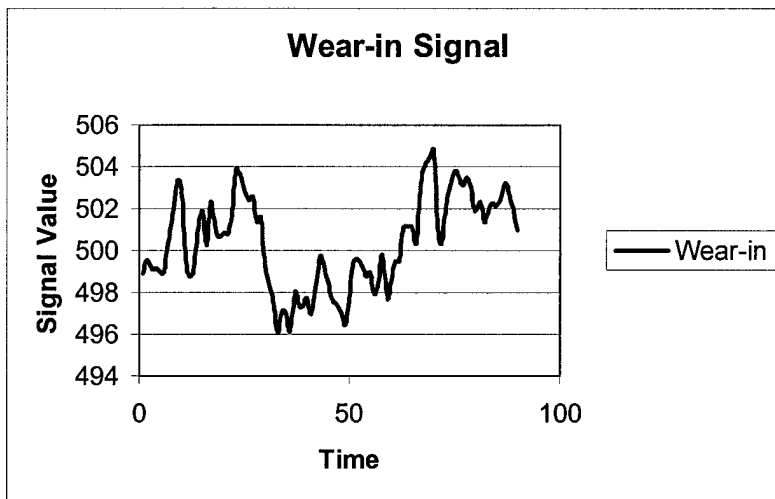


Figure 6. Wear-in Signal

The steady state signal converges around 500. The signal does show correlation, and the intermittent spikes are most likely “wear-in events”. The flight portion of the signal is built after the wear-in portion and is pictured in Figure 7.

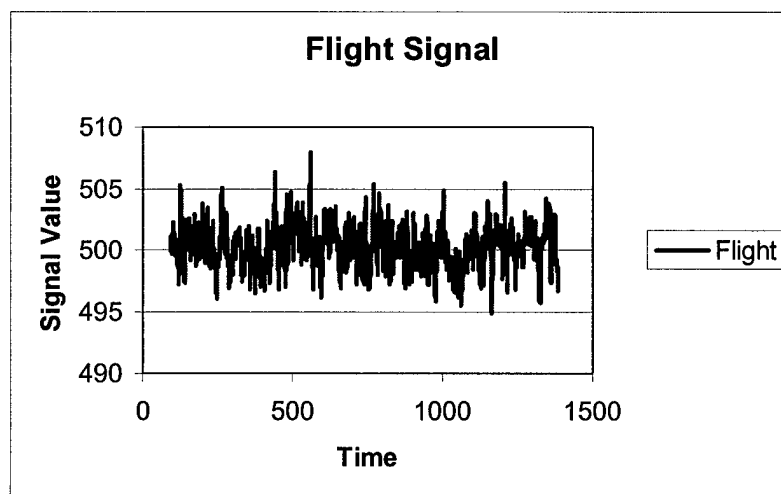


Figure 7. Flight Signal

The flight portion of the signal clearly shows some peaks that represent the “flight condition events”. The flight signal begins at time 91 and finishes at 1385. The final piece of the sensor signal is the failure portion of the signal shown in Figure 8.

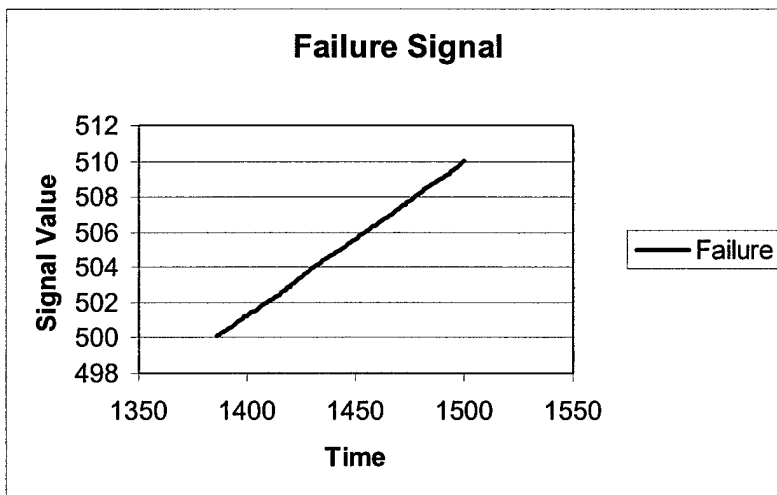


Figure 8. Failure Signal

The failure portion of the signal starts at 1386 and the part completely fails at time 1500. The Signal Generator algorithm is written such that at failure the sensor reading is 510. There is randomness in the failure signal, but it is difficult to distinguish because the signal is incremented in such small steps at each time unit. Figure 9 shows the complete signal put together by the Signal Generator.

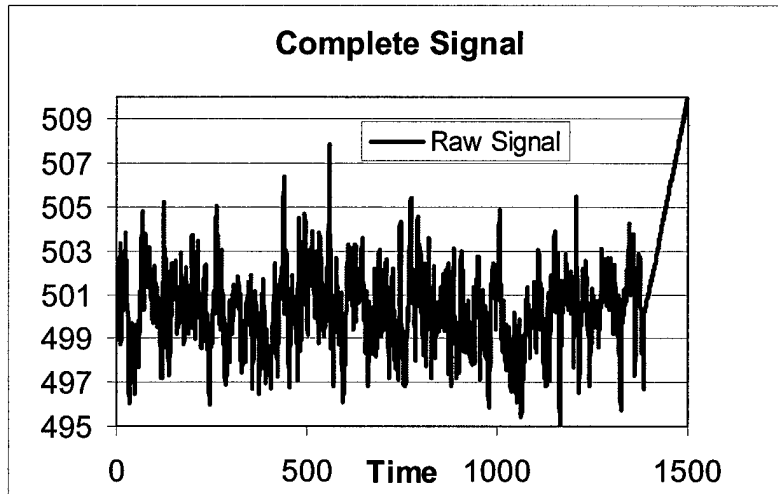


Figure 9. Complete Signal

Figure 9 clearly shows why the signal needs to be batched prior to entry into the neural network. The signal is very noisy and would lead to poor classification accuracy if it were entered as is. Batching the signal smoothes it out which leads to better results. The batched signal essentially just averages the signal for some amount of time. Later in this chapter a detailed discussion of the batch size is presented, but for illustration purposes Figure 10 shows the effect of batching using a batch size of 10. The batched signal covers the transient effects and is definitely smoother than the raw signal.

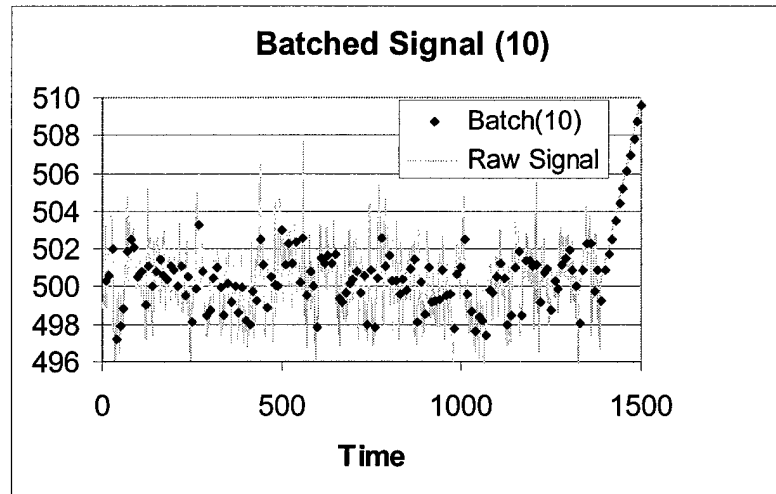


Figure 10. Batched Signal

Neural Network Analysis

To train the neural network 30 replications of each component life (Table 4) were run using a batch size of 5, 10, and 20. As mentioned in Chapter 3, the raw data is

Table 4. Component Lifetimes

Component Lifetimes									
300	700	1100	1500	2100	2700	3700	4900	7200	14100

predominantly healthy which could inadvertently cause the network to predict every batch as healthy. To prevent this condition, approximately the same number of healthy batches and failing batches were used. As an illustration consider the 30 replications using batch size 20. This results in 57,450 total signal batches, 51,777 healthy batches and 5,673 failing batches. To get a more equal number of healthy and failing batches, the 51,777 healthy batches are assigned Uniform (0, 1) random variables. The batches are sorted on the random variable and only those variables with random variables less than 0.12 are kept. The result is 6,190 healthy batches and 5,673 failing batches are used to train and validate the neural network.

Once the healthy and failing batches are selected for input into the neural network, they must be divided into training and validation data. As mentioned in Chapter 3 the data set is broke into 75% training and 25% validation. Following through on illustration above, the training set has 4,662 healthy batches and 4,304 failing batches. The validation set has 1,528 healthy batches and 1,369 failing batches. Internally SNNAP further partitions the training data set into a training set and a training test set for each epoch. This is accomplished by using a 2/3 to 1/3 ratio.

Using a batch size of 20 the neural network trained for 160 epochs and the minimized root mean square error is 0.391. Each epoch consists of passing the training data through the neural network and comparing the network predictions to the actual predictions. After each epoch, the neural network node weights are adjusted, the training and training-test data sets redefined, and then the training data is passed through the network. In the instance of batch size 20 data this iterative process occurred 160 times. One indication of how well the neural network classifies is by generating a confusion

matrix. SNNAP will generate a confusion matrix for the training data and the training test data. An Excel spreadsheet is used to build the confusion matrix for the validation data. The confusion matrices are given in Figure 11.

Training Confusion Matrix			
Actual	Predicted		
		healthy	failing
	healthy	3101	7
	failing	1044	1825
Classification Accuracy		82.42%	

Training-test Confusion Matrix			
Actual	Predicted		
		healthy	failing
	healthy	1553	1
	failing	536	899
Classification Accuracy		82.03%	

Validation Confusion Matrix			
Actual	Predicted		
		healthy	failing
	healthy	1526	2
	failing	529	840
Classification Accuracy		81.67%	

Figure 11. Trained Neural Network Confusion Matrices

The confusion matrices show how accurately the neural network classifies the signal batches. The classification accuracies listed in Figure 11 are for 30 replications of each component life, so the network has been trained over the entire range of component lifetimes.

To build the confusion matrix an assignment rule is used to classify the networks output. The neural network output (healthy and failing) ranges in value between 0 and 1. SNNAP uses a classification rule of 0.5 to generate the training and training test confusion matrices. The 0.5 cutoff equates to a batch signal value of approximately 503.878. Batch values above 503.878 are predicted as failing and values below that are

predicted as healthy. For example, if for a given batched signal the network output for healthy equals 0.3, then healthy is assigned the value 0 and failing is assigned 1.

Alternatively, if the prediction is 0.6, then healthy is assigned the value 1 and failing is assigned 0. The network's assignment is then compared to the actual component state.

Since the validation confusion matrix is built in Excel, the assignment rule can be changed to see how that impacts the networks classification accuracy. More about this will be discussed later in the chapter.

The first row of the confusion matrices can be used to classify the false alarm rate. This row represents all of the actual healthy batches. The batches that the network incorrectly predicts as failing but are in fact healthy are in effect false alarms. For example, in the validation confusion matrix there are 2 false alarm batches. There are two ways to interpret this data. First, of the 1528 healthy batches only 2 were incorrectly classified which is less than 0.5% of the batches. This would appear to be a good false alarm rate. However, the second way of looking at this is that 2 false alarms occurred in a fixed number of replications. For this initial training data set the number of replications is difficult to determine because the datasets have been broken into several pieces. When the trained neural network is used to classify 30 replications of a given component life it is not uncommon to see up to 10 false alarm batches out of 30 replications or 33% false alarms. Clearly the number of false alarms needs to be carefully defined and a classification rule used to accurately represent the data.

The second row of the confusion matrices represents all of the actual "failing" batches and can be used to classify the failure detection times. The high incidence of

healthy predictions for actual failing components signifies that there is a time delay in detecting failure.

Early Failure Detection Time versus False Alarm Tradeoff

The preceding confusion matrices are useful for illustrating a tradeoff in modeling PHM. Clearly, the goal of PHM is to minimize false alarms and to allow for the earliest possible detection of component failure. Unfortunately, these two goals are linked. As the number of false alarms is decreased, failures are detected later in the component life. There are two possible alternatives for balancing this tradeoff. The first is to change the batch size, and the second is to change the neural network assignment rule.

Batch Size Study. Since batching the signal smooths it, one method of decreasing the number of false alarms is to increase the batch size. As the batch size gets larger the batch signal is less affected by transient conditions, decreasing the variation in the signal and the number of false alarms. The drawback to this approach is that for short component lifetimes a larger batch size pushes out failure detection time.

Figure 12 shows how the batch size can be used to smooth the signal by comparing a batch size of 20 to a batch size of 10 of the same raw signal. The batch values of the batches composed of 20 signals is affected less by the transient events than the batches composed of 10 signals. The smoother signal leads to fewer healthy batches being labeled as failing.

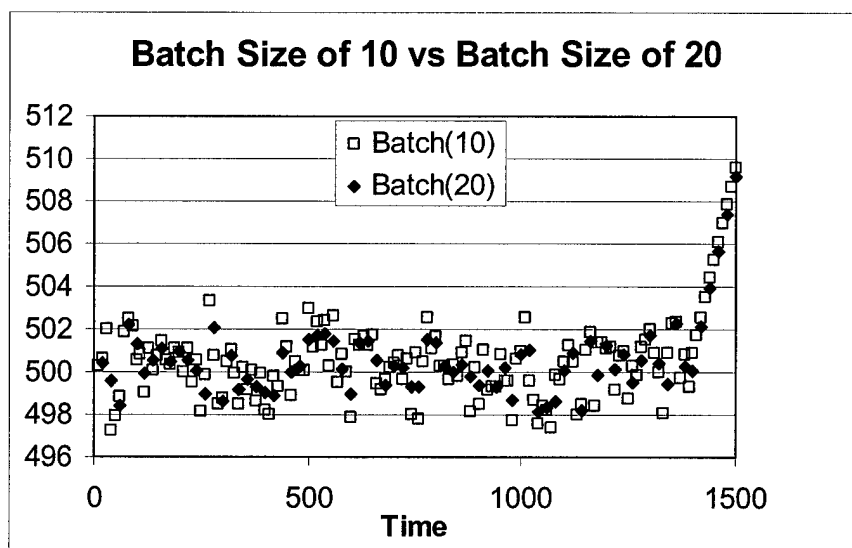


Figure 12. Batch Size of 10 vs Batch Size of 20

Figure 13 shows that a smaller batch size leads to a noisier signal that will increase the number of false alarms.

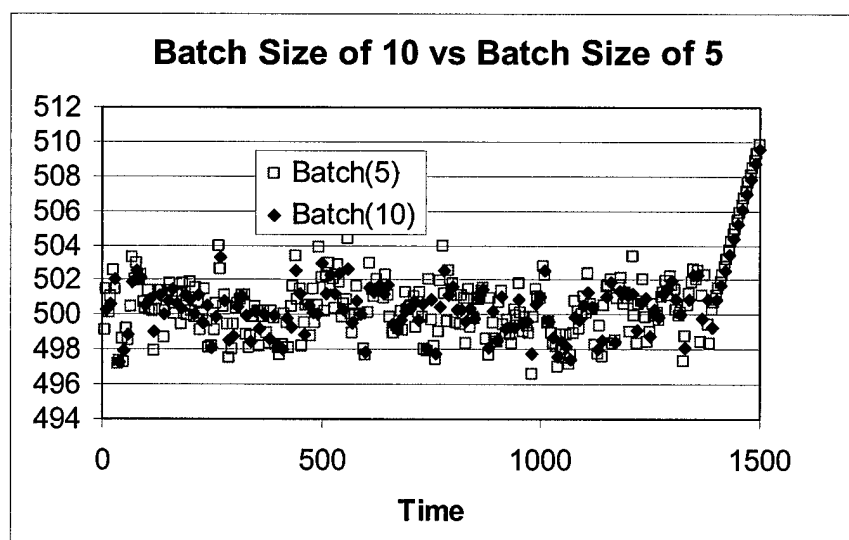


Figure 13. Batch Size 10 versus Batch Size 5

The impact of batch size on false alarm rate can be clearly seen in Table 5. The percentage under each life column represents the proportion of time that a false alarm

Table 5. Batch Size 10 and 20 False Alarm Rates

False Alarm Rate										
Life	300	700	1100	1500	2100	2700	3700	4900	7200	14100
Batch Size 20	2.00%	4.44%	5.76%	7.78%	10.89%	11.56%	14.22%	19.11%	26.67%	46.22%
Batch Size 10	14.67%	22.67%	36.44%	41.78%	54.44%	64.89%	74.44%	85.33%	92.67%	99.78%
t-test p-value	1.22E-07	8.02E-08	2.53E-09	6.40E-10	2.51E-14	3.29E-16	6.55E-19	3.18E-21	8.40E-16	1.22E-11

would occur for that component life.

The percentages were found by performing 15 macro-replications at each component life. Each macro-replication consists of 30 replications. For example, looking at the component life of 1100, a total of 450 replications were run of the Signal Generator and then put into the neural network. If during the first set of 30 replications the neural network predictions led to 5 false alarms then the false alarm rate for that replication would be 5/30 or 16.67%. The remaining 14 sets of 30 replications would similarly be analyzed and then a mean value from the 15 macro-replications can be derived. For a batch size of 10 this mean value is 36.44% and for a batch size of 20 the value is 6.36%. For this analysis any one batch misclassified as “failing” leads to a false alarm. Whether or not this is appropriate is discussed later.

While increasing the batch size is advantageous for decreasing the number of false alarms, it has an adverse affect on detecting failure time for short component lives. For example, using the Signal Generator to represent a component with life 300, the failure signal on average will begin at time 270. Using a batch size of 20, only two

batches will be labeled as failing, whereas a batch size of 10 allows three batches of failure data. As a result the batch size of 10 will detect failure earlier than the batch size of 20. Table 6 shows the affect on batch size for this experiment only impacts the failure detection time for lifetimes of 300 and 700. Beyond time 700, there is no statistical difference ($\alpha = 0.05$) between the batch size 20 and batch size 10 average failure detection times. The first row of Table 6 has the component lifetimes. The second and third rows show the average failure detection time using a batch size of 20 and a batch size of 10 respectively. As an example consider component life 700. Across 30 replications the average detection time for the batch size 20 data is 675, while the average detection time for the batch size 10 data is 669.

Table 6. Batch Size Affect on Failure Detection Time

Failure Detection Time										
Life	300	700	1100	1500	2100	2700	3700	4900	7200	14100
Batch Size 20	300	675	1054	1425	1980	2542	3495	4594	6764	13257
Batch Size 10	292	669	1047	1424	1975	2551	3486	4617	6773	13205
Benefit	100%	20%	13%	2%	4%	-6%	5%	-8%	-2%	6%
Significant Diff	Yes	Yes	No	No	No	No	No	No	No	No

The benefit measures the benefit of using a batch size of 10 versus a batch size of 20. Sticking with a component life of 700, the lead-time before failure using a batch size of 10 is 31 (700-669). Similarly the lead-time before failure using a batch size of 20 is 25 (700-675). The percent benefit is then (31-25)/31 or 0.20 (20%). The final row shows whether the difference in detection times is statistically different using $\alpha = 0.05$. The Wilcoxon Signed Rank Test is used to test the first four component lifetimes, because the results are not normally distributed. Beyond a lifetime of 1500, a two-sample t-test

(unequal variance) is used to test the difference. Using a batch size of 5 doesn't improve the average detection time for component lifetimes 300 and 700 and is actually statistically worse beyond 700 compared to batch size of 10.

Clearly batch size is very influential in minimizing the number of false alarm batches and in determining the earliest possible failure detection. For the component lifetimes used in this experiment a batch size of 20 is most advantageous in minimizing false alarms. To insure the earliest possible failure detection time a batch size of 10 is best, although the batch size 20 results are not statistically different beyond a lifetime of 700. Based on these results the batch size 20 results are carried forward for the remaining analysis.

Neural Network Assignment Rule. The other option for managing the balance between false alarms and failure detection time is to modify the neural network assignment rule. For the analysis presented above the neural network classified a signal batch as "healthy" if the healthy prediction is greater than 0.5; similarly a signal batch is "failing" if the failing prediction is greater than 0.5. Any given batch will only satisfy one of the above conditions.

One method of looking at the sensitivity of the neural network output to the classification rule is to build a Receiver Operating Characteristic (ROC) curve. A ROC curve shows the relationship between some parameter and the output of the neural network. Figure 14 below shows a ROC curve for a component with life 1500.

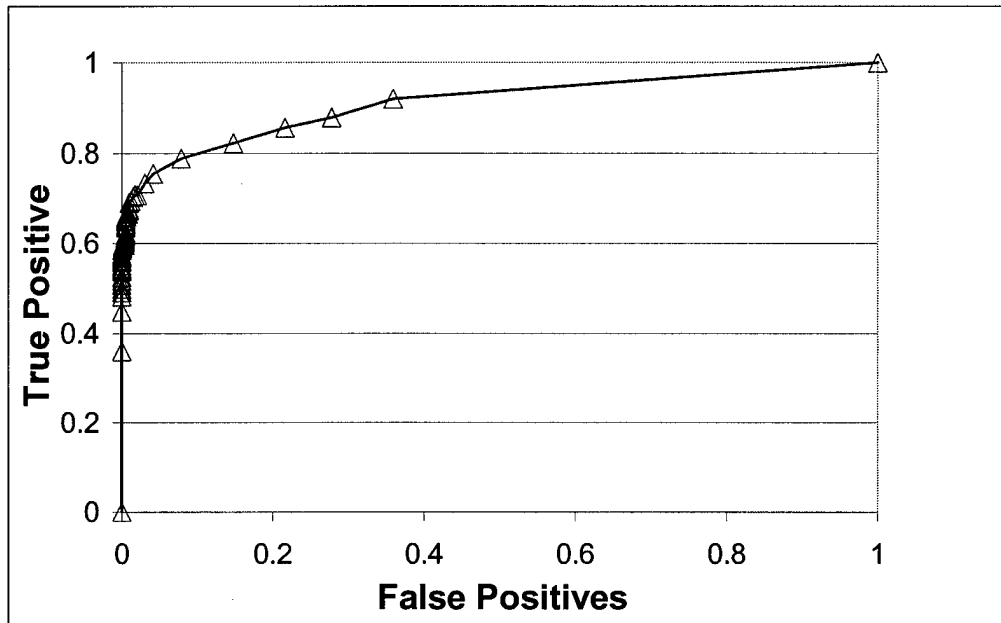


Figure 14. ROC Curve Example

To build the ROC curve the network classification rule is varied from healthy equal zero and failing equal one to healthy equal one and failing equal zero. The false positive axis corresponds to the first row of the confusion matrix, and the true positive axis corresponds to the second row of the confusion matrix. The bottom left corner of the ROC curve represents one extreme of trying to minimize the number of false positives (false alarms). In order to achieve this goal, the ability of the network to predict failing components is compromised. The other extreme (upper right corner) is to error towards predicting when a component is failing. For this thesis application, operating in this region insures that the network detects a failing component immediately.

Figure 15 makes the transition from the ROC curve to false alarm rates, and failure detection time. Figure 15 includes the average false alarm rate and average failure

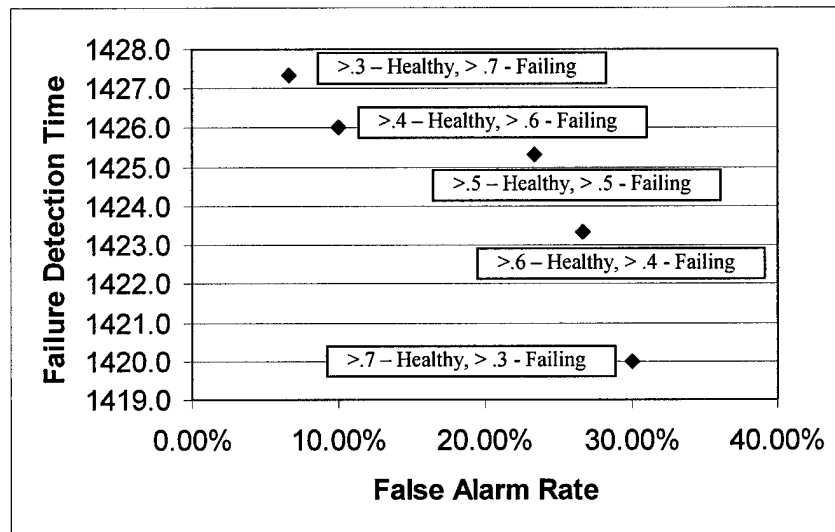


Figure 15. Affect of Changing Classification Rule

detection time for five different assignment rules. Starting from the upper left hand corner of the graph, the classification rule is ranged from 0.3 healthy, 0.7 failing to 0.7 healthy, 0.3 failing. When the classification rule is healthy greater than 0.3, the neural network prediction has a low false alarm rate. As that classification rule is increased to 0.7, the number of false alarms increases. The tradeoff again is the failure detection time. When the network classification errors toward healthy, the failure detection time is later than when the network classification errors towards failing.

Over the full range of the classification rule, the failure detection time changes by 10% (1420 to 1428). The difference in the false alarm rate is more substantial, varying

between 10% and 30%. These results are for only 30 replications (versus 15 macro-replications), so the false alarm rate results are not actually this dramatic. Table 5 shows that the actual false alarm rate for a component life of 1500 is 7.78% (.5, .5) versus the approximate 22% shown above. Since the false alarm rate for the batch size 20 data is not too bad using the .5, .5 classification, the remainder of the analysis will be performed using that rule. Once ALSim is modified, sensitivity analysis can be performed by varying the classification rule to determine the optimum setting.

Failure Detection Time and False Alarm Rate Results

After the Signal Generator produced the initial sensor signals, and the neural network was trained to predict healthy and failing signal batches, the failure detection time data and false alarm data can be obtained. The failure detection data is obtained by running 30 replications of each component life through the neural network. Thirty replications are sufficient to get an average failure detection time and to determine the spread of the detection times. The false alarm data is generated as discussed above. Included below is some discussion about what false alarm strategy is appropriate.

Failure Detection Time. The failure detection data collection was done for batch sizes of 5, 10, and 20. The average failure detection times for batch size of 10 and 20 were presented in Table 6. The average failure detection time using a batch size of 5 is worse than the batch size of 10. The batch size 5 results are worse because the batches are influenced too much by the transient events. The results from Table 6 show that there

is virtually no difference in the average failure detection time for the batch size 10 and batch size 20 data except early in the component life.

The next step in building the PHM failure detection capability is to determine the spread of the data. The failure detection times for all 30 replications were entered into *Arena's Input Analyzer*, to build a distribution of the failure detection times. The highest-rank distribution for both batch sizes is shown in Table 7. The Arena p-values

Table 7. Failure Time Distribution

Batch Size 10				Batch Size 20			
Life	Dist/Parameters	Arena p-value	Shapiro-Wilk	Life	Dist/Parameters	Arena p-value	Shapiro-Wilk
300	Tria(280, 290, 301)	< .005	< .001	300	NA		
700	Tria(660, 668, 681)	0.125	< .001	700	Tria(660, 681, 701)	0.0206	
1100	Tria(1010, 1060, 1070)	> 0.75	0.09	1100	Tria(1020, 1060, 1080)	> 0.75	
1500	Tria(1390, 1410, 1470)	0.236	0.12	1500	Tria(1380, 1440, 1460)	0.3	
2100	Norm(1980, 28)	> 0.15	0.33	2100	Norm(1980, 28.3)	> 0.15	0.17
2700	Norm(2550, 33.3)	> 0.15	0.76	2700	Norm(2540, 28.9)	> 0.15	0.28
3700	Norm(3490, 44.9)	> 0.15	0.23	3700	Norm(3500, 51)	> 0.15	0.20
4900	Norm(4620, 58.5)	> 0.15	0.87	4900	Norm(4590, 46.8)	> 0.15	0.42
7200	Norm(6770, 83.6)	> 0.15	0.15	7200	Norm(6760, 97.6)	> 0.15	0.05
14100	Norm(13200, 162)	> 0.15	0.90	14100	Norm(13300, 153)	> 0.15	0.68

use the Chi Squared test for the triangular distributions and the Kolmogorov-Smirnov value for the normality of the data. The Shapiro-Wilk test is included in SAS JMP. Five or six histogram intervals were used to visually analyze the data, based on Sturge's Rule. Beyond a lifetime of 1500, the data is clearly normally distributed for both batch sizes. The component lifetimes at or prior to 1500 have failure detection times that are too bunched together to be normally distributed and are better fit using a triangular distribution. Two example histograms are presented in Figure 16 and Figure 17.

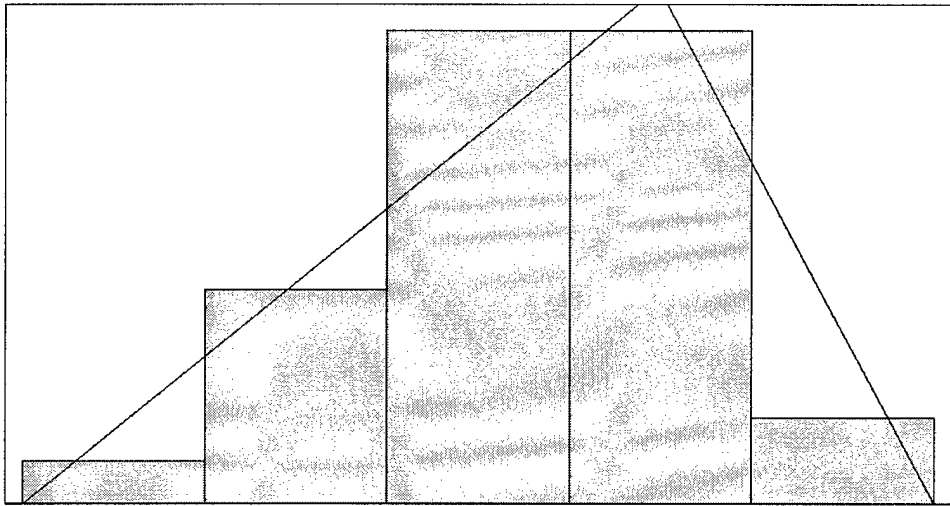


Figure 16. Failure Time Distribution for 1500 Life

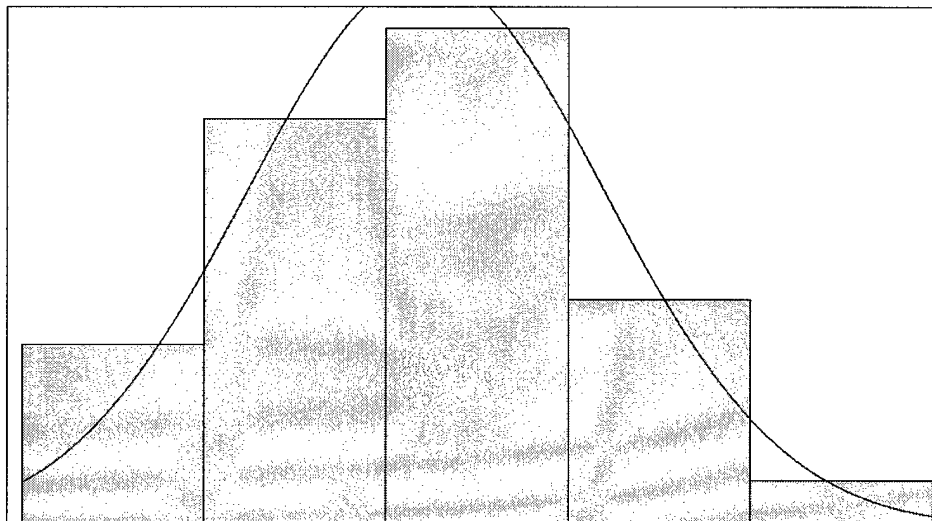


Figure 17. Failure Time Distribution for 3700 Life

False Alarm Rate. Unlike the failure detection time results, the false alarm rates for the different batch sizes are quite different. Table 5 presented earlier in the chapter clearly shows a difference in the number of false alarms for a batch size of 10 and a batch size of 20 (batch size of 5 results in higher false alarm rates than 10). In modeling the false alarm rates the goal is to be as accurate as possible. For the results presented in Table 5 a false alarm consists of at least one misclassified batch by the neural network. This may not be the best “rule” to use in defining a false alarm, but provides a reasonable starting point.

The false alarm data from the batch size 10 data indicates a different definition of false alarm may be needed. One possibility is that a false alarm could be defined as two false alarm batches in a row. This makes sense, however the Signal Generator algorithm is built such that the transient event conditions are truly random. Out of all the replications performed, the false alarms for the batch size 20 data are roughly uniformly distributed (using *Arena Input Analyzer*). As a result there are not any occasions where two sequential batches are falsely predicted as failing. The batch size 10 data are distributed according to a beta distribution, but similarly there are no sequential batches. Another possible “rule” for assigning false alarms would be to classify a false alarm as any replication that had two or more false alarm batches at any point during the replication. This leads to lower false alarm rates, but does not have any operational or logical basis.

Based on the results from the failure detection study and the false alarm study, the batch size 20 data was used in ALSim to modify the PHM component. The failure detection time for the batch size 20 data is very similar to the batch size 10 data, but has

more reasonable false alarm rates. For this research effort a false alarm will be classified if any batch in a replication is falsely classified as failing by the neural network. That means the false alarm probabilities in Table 5 will be used. Perhaps further research can be done in this area to discover a more appropriate method.

ALSim Inputs

The Signal Generator and neural network analysis were done to build a distribution of failure detection times, and add false alarm capability to ALSim. The changes to ALSim to incorporate these changes are straightforward. The only Java class in ALSim that needs to be modified is **PHM**. Currently, in the **PHM** class a detection factor is used to determine when PHM detects a component failure. The failure detection time distribution replaces this detection factor, and a simple uniform random draw adds the false alarm capability.

Failure Detection Time. ALSim currently logs three notional line replaceable units (LRUs), for which the mean time between failures 433.7 hours, 833 hours, and 952.31 hours. The Signal Generator was used to build a signal for a component with mean time between failure of 3000. Realistically the time units on the Signal Generator are seconds or minutes, not hours. This difference in time requires a logical and flexible method for scaling the Signal Generator results into ALSim.

One possible solution to the scaling problem is scale the results using the cumulative exponential distribution. Table 8 shows how the scaling would look for all

Table 8. Scaling Lifetime

Cumulative Dist	0.10	0.21	0.31	0.39	0.50	0.59	0.71	0.80	0.91	0.99
Signal Gen Life (3000)	300	700	1100	1500	2100	2700	3700	4900	7200	14100
LRU 1 (952.31)	100	220	340	490	675	875	1150	1550	2250	4575
LRU 2 (833)	85	190	300	430	590	775	1000	1350	2000	4000
LRU 3 (433.7)	50	100	160	225	300	400	525	700	1000	2000

three LRUs. The mean time between failure for LRU 1 is 952.31 hours. If the random failure draw results in a failure of 875 hours, the failure detection time and probability of false alarm would coincide with the Signal Generator lifetime of 2700. The difficulty in using this approach is that you assume that everything in the Signal Generator, batch size study, and neural network analysis scales from 3000 seconds or minutes to 952.31 hours.

Another approach for solving the scaling problem is to look for characteristics or trends in the Signal Generator data for different component lifetimes that can easily be implemented in ALSim. Generalized trends are sufficient for developing a meaningful PHM modeling approach, and don't require such strict assumptions as presented above. Table 9 shows the average failure detection time as a percent of the component lifetime for the batch size 20 results and batch size 10 results. Although there is a difference at

Table 9. Failure Detection Time Trend

Life	300	700	1100	1500	2100	2700	3700	4900	7200	14100	Average
Failure Detection (20)	300	675	1054	1425	1980	2542	3495	4594	6764	13257	
Percent of Life	100%	96%	96%	95%	94%	94%	94%	94%	94%	94%	95%
Failure Detection (10)	292	669	1047	1424	1975	2551	3486	4617	6773	13205	
Percent of Life	97%	96%	95%	95%	94%	94%	94%	94%	94%	94%	95%

lifetime of 300 (confirmed in Table 6), beyond that the detection times are virtually the same. In fact, the mean of average failure detection time as a percent of life for both batch sizes is 95%. This average failure detection time provides a simple and flexible method for incorporation of our Signal Generator data into ALSim.

To test how robust this approach was, all the failure detection time data was converted to a percent of component life and put into *Arena Input Analyzer*. For each component life (300 to 14100), the failure detection times across 30 replications were simply divided by the component life. This yielded 300 values between 0.90 and 1.00 (90% to 100%). Putting these values into *Arena Input Analyzer* yields the following histogram (Figure 18).

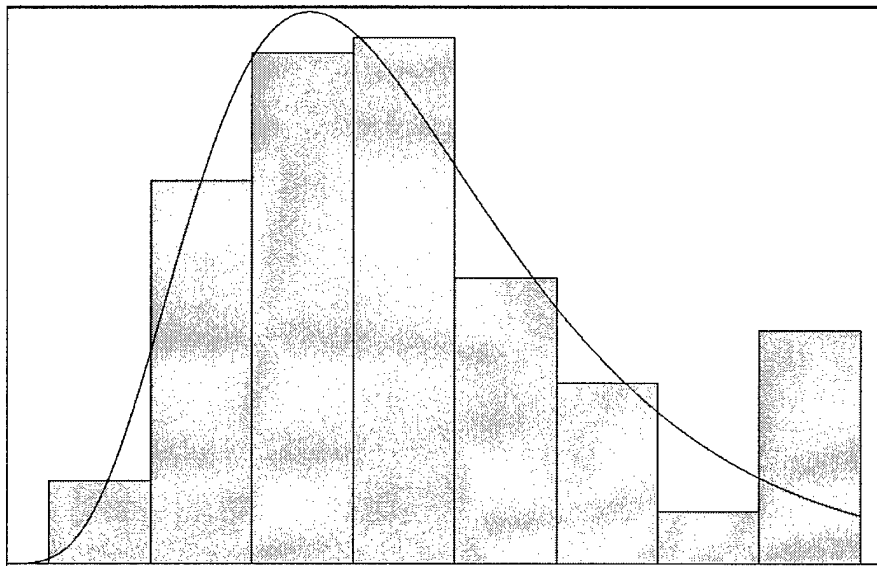


Figure 18. Failure Detection Time as Percent of Life (1)

The left side of the histogram starts with value 0.914, and the right side ends with 1.00.

The data is best fit using a lognormal distribution, and is not normally distributed because of the influence of the high percentages associated with component lifetime 300 and 700.

All of the component lifetime 300 data has failure detection time of 300, which leads to 30 instances of 1.0. Removing the component 300 and 700 data yields the following histogram (Figure 19). It is reasonable to remove these two early data points because they represent a small percentage of the actual lifetimes. The histogram is best fit with a

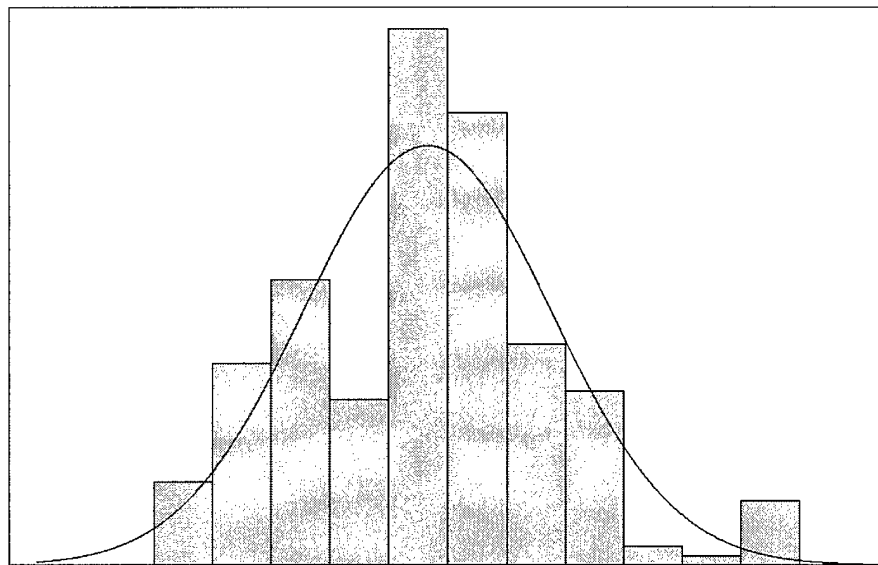


Figure 19. Failure Detection Time as Percent of Life (2)

normal distribution with mean .944 (94.4%) and standard deviation 0.0141 (1.41%). The Kolmogorov-Smirnov p-value for the distribution being normally distributed is 0.15. The interpretation of these results is that the average failure detection time is 94.4% of the

component lifetime with standard deviation of 1.41%. The failure detection time in ALSim was modified using this information.

False Alarms. The false alarm data is more difficult to generalize because it is time dependent. The longer a component is in use the greater the probability it will experience a false alarm. Scaling from the Signal Generator component lifetimes is shown in Table 10. To read Table 10, consider LRU 1 component lifetime of 400.

Table 10. False Alarm Scaling

False Alarm Rate	0.02	0.04	0.06	0.08	0.11	0.12	0.14	0.18	0.27	0.38
Signal Gen Life (3000)	300	700	1100	1500	2100	2700	3700	4900	7200	14100
LRU 1 (952.31)	100	220	340	490	675	875	1150	1550	2250	4575
LRU 2 (833)	85	190	300	430	590	775	1000	1350	2000	4000
LRU 3 (433.7)	50	100	160	225	300	400	525	700	1000	2000

The probability of a false alarm with this component lifetime is 0.08. Said differently, 8% of the time a component with this life will experience a false alarm. To add this capability to ALSim, conditional statements were added to the **PHM** class for each LRU.

ALSim Operation. The PHM component of ALSim now has a built in failure detection time distribution, and false alarm capability. The PHM component operates by first selecting a component lifetime which represents the point at which the JSF will have to be grounded because the component has completely failed. This function is unchanged from Capt Rebulanan's work. After the component lifetime is determined, a uniform (0,1) random draw is performed to determine if a false alarm will occur. The random number is compared to the false alarm rate for the component lifetime. If the random

number is less than the false alarm rate than a false alarm does occur. Another uniform random number draw ($0, 0.9 \times \text{component life}$) is used to determine the time of the false alarm. The maintenance system responds to false alarms as if it were a true failure. If the uniform $(0, 1)$ false alarm random draw is greater than the false alarm rate then the system operates until failure detection. The failure detection time is determined using a normal $(0.944, 0.0141)$ random draw and multiplying the result by the component life.

Conclusions

This chapter outlined the results of this thesis effort. Employing the methodology discussed in Chapter 3, the results led to a more accurate and realistic PHM component of ALSim. The batch size study and false alarm analysis showed that using a batch size of 20 leads to the most realistic failure detection times and false alarm rates. The impact of varying the neural network classification rule was also presented. The baseline classification of .5 healthy, and .5 failing was used to model PHM.

Using the established neural network and a batch size of 20 the Signal Generator was run to generate 450 replications for each component lifetime. The projection of the neural network onto these replications yields the failure detection time and false alarm rate information needed in ALSim. The final step was to put the failure detection time distribution and false alarm rate into ALSim. The failure detection time was implemented easily because the trend was that failure of a component was detected at approximately 95% of its life. The false alarm rate had to be scaled from the Signal

Generator lifetimes to the ALSim LRU lifetimes. This was accomplished using the exponential distribution.

V. Conclusions and Recommendations

The objective of this thesis research was to develop a PHM modeling methodology to employ in the JSF ALSim model. It's important to remember that once the JSF is operational the PHM modeling implementation will be a given. Data will exist on the PHM capability, but in the meantime this research provides a useful tool for decision makers trying to make a decision on a system that is not yet built. This research provides insight into how PHM operates, and the sensitivity of PHM to changing aircraft conditions. It is important to properly model and understand the PHM system being used in the JSF, because it is the enabling technology of the autonomic logistics system.

This thesis effort began with an extensive literature review. The purpose of the literature review was to find information that could be used to model PHM. Journal articles on PHM, predictive maintenance strategies, data analysis and simulation techniques were found and presented. The literature review provided a framework to solve the problem – the methodology. The methodology developed to model PHM involved analyzing a prognostic sensor signal with a neural network to determine when the component associated with the prognostic would fail. Additionally, the neural network enabled false alarm data to be determined. The preferred approach to build and train the neural network was to use actual JSF prognostic data. Unfortunately that data is not available. To overcome this, an interactive model was built in Java that generates a prognostic signal based on user input.

A key part of the methodology was finding a way to enter the signal data into the neural network. The raw signal was purposefully transient in nature to be as realistic as possible. Entering the raw signal would have led to poor classification accuracy, resulting in an unreasonable number of false alarms and unreasonable failure detection times. A batching strategy was developed and implemented to overcome the transient nature of the signal.

After the methodology, the results were presented. The impact of batch size on false alarm rate and failure detection time was shown. The conclusion was that the batch size must be chosen such that it sufficiently masks the transient effects. A batch size too small produces unreasonable false alarm rates, and a batch size too large delays the detection of an impending failure. Another important factor in determining appropriate false alarm rate and failure detection times is the neural network assignment rule. The neural network had to classify a batched signal as healthy or failing. If the assignment rule is implemented to minimize false alarms the failure detection time increases. Conversely, the network can maximize the lead-time in detecting a failure, but the false alarm rate will be higher. The best method to optimize the neural network classification is to use ALSim to differentiate between the two strategies.

For the Signal Generator settings used in this analysis a very simple failure detection time capability was added to ALSim. On average a failure is detected at 95% of its life. The analysis clearly showed that false alarms are time dependent and need to be modeled as such. Basically the longer a component operates the higher its probability of experiencing a false alarm. The ALSim modifications were minor to implement the failure detection time. Since the false alarm data is time dependent, the analysis results

from the Signal Generator had to be scaled to the existing LRUs in ALSim. The ALSim false alarm modifications were then entered as conditional statements.

Further Study

This thesis effort really was a proof-of-concept demonstration. It showed that batching a prognostic signal and then building a neural network to predict whether the signal is healthy or failing is a reasonable approach to model PHM. This approach quantifies the false alarm rate and develops a distribution of failure detection times for a component. The goal in the future is to be able to implement the modeling strategy using actual JSF data.

Until the actual data becomes available, there is additional analysis that can be done to characterize PHM and ALS. Clearly this research involved many assumptions that could be analyzed using a sensitivity analysis. One example is the Signal Generator failure signal. Currently the failure portion of the signal is assumed to be linear, and starts at roughly 90% of the component life. This failure start time could be varied to look at the impact of the failure detection time; or exponential or quadratic functions could be used to approximate the failure signal. Another example of sensitivity analysis would be to run the Signal Generator at several different settings and train the neural network. The results presented in this effort are very specific and probably are not robust to changing input conditions.

Another possible follow-on effort could look at how statistical process control (SPC) could be used to model PHM. An SPC approach could look at building limits around a prognostic signal to detect failure.

Appendix A. Signal Generator Code

```
/* Capt Mike Malley      **
** GOR01M                **
**                      **
** Last modified: 27 Dec 00 */

/*
    This simple extension of the java.awt.Frame class
    contains all the elements necessary to act as the
    main window of an application.
*/

import java.awt.*;
import javax.swing.JTextArea;
import java.lang.String;
import java.util.*;
import java.io.*;

public class JsGui extends Frame
{
    public JsGui()
    {
        // This code is automatically generated by Visual Cafe when you add
        // components to the visual environment. It instantiates and initializes
        // the components. To modify the code, only use code syntax that matches
        // what Visual Cafe can generate, or Visual Cafe may be unable to back
        // parse your Java file into its visual environment.

        // {INIT_CONTROLS
        setLayout(null);
        setSize(875,572);
        setVisible(false);
        openFileDialog1.setMode(FileDialog.LOAD);
        openFileDialog1.setTitle("Open");
        // $$ openFileDialog1.move(24,420);
        labelTitle.setText("JSF PHM Signal Generator");
        labelTitle.setAlignment(java.awt.Label.CENTER);
        add(labelTitle);
        labelTitle.setBackground(java.awt.Color.blue);
        labelTitle.setForeground(java.awt.Color.white);
        labelTitle.setFont(new Font("Dialog", Font.BOLD, 16));
        labelTitle.setBounds(36,12,231,49);
        textExplain.setEditable(false);
        textExplain.setText("The JSF PHM Signal Generator builds a PHM sensor signal based on the selected
settings.");
        add(textExplain);
        textExplain.setBounds(36,84,228,60);
        textMean.setText("100");
        add(textMean);
        textMean.setBounds(204,156,63,24);
        textLife.setText("300");
        add(textLife);
        textLife.setBounds(204,192,63,24);
        labelMean.setText("Signal Nominal Mean");
        add(labelMean);
        labelMean.setBounds(48,156,136,19);
        labelVariance.setText("Component Life");
        add(labelVariance);
        labelVariance.setBounds(48,192,136,19);
        panelWearin.setLayout(null);
    }
}
```

```

add(panelWearin);
panelWearin.setBounds(336,24,348,97);
labelWearin.setText("Signal Sensitivity to Component Wearin");
labelWearin.setAlignment(java.awt.Label.CENTER);
panelWearin.add(labelWearin);
labelWearin.setBackground(new java.awt.Color(255,128,0));
labelWearin.setForeground(java.awt.Color.white);
labelWearin.setFont(new Font("Dialog", Font.BOLD, 14));
labelWearin.setBounds(0,0,348,24);
weartimeScrollbar.setBlockIncrement(3);
panelWearin.add(weartimeScrollbar);
weartimeScrollbar.setBounds(0,48,153,21);
wearrateScrollbar.setBlockIncrement(5);
panelWearin.add(wearrateScrollbar);
wearrateScrollbar.setBounds(192,48,153,21);
labelWearTL.setText("0");
panelWearin.add(labelWearTL);
labelWearTL.setBounds(0,24,45,26);
labelWearTU.setText("15");
labelWearTU.setAlignment(java.awt.Label.RIGHT);
panelWearin.add(labelWearTU);
labelWearTU.setBounds(108,24,45,26);
labelWearRL.setText("0");
panelWearin.add(labelWearRL);
labelWearRL.setBounds(192,24,45,26);
labelWearRU.setText("50");
labelWearRU.setAlignment(java.awt.Label.RIGHT);
panelWearin.add(labelWearRU);
labelWearRU.setBounds(300,24,45,26);
label5.setText("3");
label5.setAlignment(java.awt.Label.CENTER);
panelWearin.add(label5);
label5.setBackground(java.awt.Color.lightGray);
label5.setBounds(48,24,45,26);
label6.setText("15");
label6.setAlignment(java.awt.Label.CENTER);
panelWearin.add(label6);
label6.setBackground(java.awt.Color.lightGray);
label6.setBounds(240,24,45,26);
label7.setText("Wearin time as % of life");
label7.setAlignment(java.awt.Label.CENTER);
panelWearin.add(label7);
label7.setBounds(0,72,144,24);
label8.setText("Wearin occurrence rate (%)");
label8.setAlignment(java.awt.Label.CENTER);
panelWearin.add(label8);
label8.setBounds(192,72,156,24);
panel1.setLayout(null);
add(panel1);
panel1.setBackground(java.awt.Color.white);
panel1.setBounds(336,156,348,97);
label9.setText("Signal Sensitivity to Changing Flight Conditions");
label9.setAlignment(java.awt.Label.CENTER);
panel1.add(label9);
label9.setBackground(java.awt.Color.red);
label9.setForeground(java.awt.Color.white);
label9.setFont(new Font("Dialog", Font.BOLD, 14));
label9.setBounds(0,0,348,24);
flightScrollbar.setBlockIncrement(5);
panel1.add(flightScrollbar);
flightScrollbar.setBounds(60,48,201,21);
labelFlightL.setText("0");
panel1.add(labelFlightL);
labelFlightL.setBounds(60,24,45,26);
labelFlightU.setText("50");
labelFlightU.setAlignment(java.awt.Label.RIGHT);
panel1.add(labelFlightU);
labelFlightU.setBounds(216,24,45,26);

```

```

label15.setText("10");
label15.setAlignment(java.awt.Label.CENTER);
panel1.add(label15);
label15.setBackground(java.awt.Color.lightGray);
label15.setBounds(132,24,45,26);
label17.setText("Adverse flight condition occurrence rate (%)");
label17.setAlignment(java.awt.Label.CENTER);
panel1.add(label17);
label17.setBounds(36,72,252,24);
panel2.setLayout(null);
add(panel2);
panel2.setBounds(336,288,348,97);
label18.setText("PHM Failure Prediction");
label18.setAlignment(java.awt.Label.CENTER);
panel2.add(label18);
label18.setBackground(new java.awt.Color(0,128,0));
label18.setForeground(java.awt.Color.white);
label18.setFont(new Font("Dialog", Font.BOLD, 14));
label18.setBounds(0,0,348,24);
controlScrollbar.setBlockIncrement(1);
panel2.add(controlScrollbar);
controlScrollbar.setBounds(0,48,153,21);
failureScrollbar.setBlockIncrement(5);
panel2.add(failureScrollbar);
failureScrollbar.setBounds(192,48,153,21);
labelControlL.setText("1");
panel2.add(labelControlL);
labelControlL.setBounds(0,24,45,26);
labelControlU.setText("6");
labelControlU.setAlignment(java.awt.Label.RIGHT);
panel2.add(labelControlU);
labelControlU.setBounds(108,24,45,26);
labelFailureL.setText("0");
panel2.add(labelFailureL);
labelFailureL.setBounds(192,24,45,26);
labelFailureU.setText("20");
labelFailureU.setAlignment(java.awt.Label.RIGHT);
panel2.add(labelFailureU);
labelFailureU.setBounds(300,24,45,26);
label23.setText("3");
label23.setAlignment(java.awt.Label.CENTER);
panel2.add(label23);
label23.setBackground(java.awt.Color.lightGray);
label23.setBounds(48,24,45,26);
label24.setText("20");
label24.setAlignment(java.awt.Label.CENTER);
panel2.add(label24);
label24.setBackground(java.awt.Color.lightGray);
label24.setBounds(240,24,45,26);
label25.setText("Control limit settings ");
label25.setAlignment(java.awt.Label.CENTER);
panel2.add(label25);
label25.setBounds(0,72,144,24);
label26.setText("Variability in failure start");
label26.setAlignment(java.awt.Label.CENTER);
panel2.add(label26);
label26.setBounds(180,72,168,24);
textMTBF.setText("3000");
add(textMTBF);
textMTBF.setBounds(204,228,63,24);
labelMTBF.setText("Component MTBF ");
add(labelMTBF);
labelMTBF.setBounds(48,228,136,19);
panel3.setLayout(null);
add(panel3);
panel3.setBackground(java.awt.Color.yellow);
panel3.setBounds(36,276,228,101);
button1.setLabel("Run Simulation");

```

```

panel3.add(button1);
button1.setBackground(java.awt.Color.lightGray);
button1.setBounds(48,12,134,50);
labelReps.setText("Number of Replications");
panel3.add(labelReps);
labelReps.setBounds(12,72,136,19);
textReps.setText("30");
panel3.add(textReps);
textReps.setBackground(java.awt.Color.white);
textReps.setBounds(156,72,63,24);
setTitle("JSF Signal Generator");
//}}

//{{INIT_MENU
menu1.setLabel("File");
menu1.add(newMenuItem);
newMenuItem.setEnabled(false);
newMenuItem.setLabel("New");
newMenuItem.setShortcut(new MenuShortcut(java.awt.event.KeyEvent.VK_N,false));
menu1.add(openMenuItem);
openMenuItem.setLabel("Open...");
openMenuItem.setShortcut(new MenuShortcut(java.awt.event.KeyEvent.VK_O,false));
menu1.add(saveMenuItem);
saveMenuItem.setEnabled(false);
saveMenuItem.setLabel("Save");
saveMenuItem.setShortcut(new MenuShortcut(java.awt.event.KeyEvent.VK_S,false));
menu1.add(saveAsMenuItem);
saveAsMenuItem.setEnabled(false);
saveAsMenuItem.setLabel("Save As...");
menu1.add(separatorMenuItem);
separatorMenuItem.setLabel("-");
menu1.add(exitMenuItem);
exitMenuItem.setLabel("Exit");
mainMenuBar.add(menu1);
menu2.setLabel("Edit");
menu2.add(cutMenuItem);
cutMenuItem.setEnabled(false);
cutMenuItem.setLabel("Cut");
cutMenuItem.setShortcut(new MenuShortcut(java.awt.event.KeyEvent.VK_X,false));
menu2.add(copyMenuItem);
copyMenuItem.setEnabled(false);
copyMenuItem.setLabel("Copy");
copyMenuItem.setShortcut(new MenuShortcut(java.awt.event.KeyEvent.VK_C,false));
menu2.add(pasteMenuItem);
pasteMenuItem.setEnabled(false);
pasteMenuItem.setLabel("Paste");
pasteMenuItem.setShortcut(new MenuShortcut(java.awt.event.KeyEvent.VK_V,false));
mainMenuBar.add(menu2);
menu3.setLabel("Help");
menu3.add(aboutMenuItem);
aboutMenuItem.setLabel("About...");
mainMenuBar.add(menu3);
//$$ mainMenuBar.move(0,420);
setMenuBar(mainMenuBar);
//}}

//{{REGISTER_LISTENERS
SymWindow aSymWindow = new SymWindow();
this.addWindowListener(aSymWindow);
SymAction ISymAction = new SymAction();
openMenuItem.addActionListener(ISymAction);
exitMenuItem.addActionListener(ISymAction);
aboutMenuItem.addActionListener(ISymAction);
SymAdjustment ISymAdjustment = new SymAdjustment();
weartimeScrollbar.addAdjustmentListener(ISymAdjustment);
wearrateScrollbar.addAdjustmentListener(ISymAdjustment);
flightScrollbar.addAdjustmentListener(ISymAdjustment);
controlScrollbar.addAdjustmentListener(ISymAdjustment);

```

```

        failureScrollbar.addAdjustmentListener(lSymAdjustment);
        button1.addActionListener(lSymAction);
        //}}
    }

    public JsGui(String title)
    {
        this();
        setTitle(title);
    }

    /**
     * Shows or hides the component depending on the boolean flag b.
     * @param b if true, show the component; otherwise, hide the component.
     * @see java.awt.Component#isVisible
     */
    public void setVisible(boolean b)
    {
        if(b)
        {
            setLocation(50, 50);
        }
        super.setVisible(b);
    }

    static public void main(String args[])
    {
        try
        {
            //Create a new instance of our application's frame, and make it visible.
            (new JsGui()).setVisible(true);
        }
        catch (Throwable t)
        {
            System.err.println(t);
            t.printStackTrace();
            //Ensure the application exits with an error condition.
            System.exit(1);
        }
    }

    public void addNotify()
    {
        // Record the size of the window prior to calling parents addNotify.
        Dimension d = getSize();

        super.addNotify();

        if (fComponentsAdjusted)
            return;

        // Adjust components according to the insets
        setSize(getInsets().left + getInsets().right + d.width, getInsets().top + getInsets().bottom + d.height);
        Component components[] = getComponents();
        for (int i = 0; i < components.length; i++)
        {
            Point p = components[i].getLocation();
            p.translate(getInsets().left, getInsets().top);
            components[i].setLocation(p);
        }
        fComponentsAdjusted = true;
    }

    // Used for addNotify check.
    boolean fComponentsAdjusted = false;

    //{DECLARE_CONTROLS
    java.awt.FileDialog openFileDialog1 = new java.awt.FileDialog(this);

```

```

java.awt.Label labelTitle = new java.awt.Label();
java.awt.TextArea textExplain = new java.awt.TextArea("",0,0,TextArea.SCROLLBARS_NONE);
java.awt.TextField textMean = new java.awt.TextField();
java.awt.TextField textLife = new java.awt.TextField();
java.awt.Label labelMean = new java.awt.Label();
java.awt.Label labelVariance = new java.awt.Label();
java.awt.Panel panelWearin = new java.awt.Panel();
java.awt.Label labelWearin = new java.awt.Label();
java.awt.Scrollbar weartimeScrollbar = new java.awt.Scrollbar(Scrollbar.HORIZONTAL,3,2,0,17);
java.awt.Scrollbar wearrateScrollbar = new java.awt.Scrollbar(Scrollbar.HORIZONTAL,15,5,0,55);
java.awt.Label labelWearTL = new java.awt.Label();
java.awt.Label labelWearTU = new java.awt.Label();
java.awt.Label labelWearRL = new java.awt.Label();
java.awt.Label labelWearRU = new java.awt.Label();
java.awt.Label label5 = new java.awt.Label();
java.awt.Label label6 = new java.awt.Label();
java.awt.Label label7 = new java.awt.Label();
java.awt.Label label8 = new java.awt.Label();
java.awt.Panel panel1 = new java.awt.Panel();
java.awt.Label label9 = new java.awt.Label();
java.awt.Scrollbar flightScrollbar = new java.awt.Scrollbar(Scrollbar.HORIZONTAL,10,5,0,55);
java.awt.Label labelFlightL = new java.awt.Label();
java.awt.Label labelFlightU = new java.awt.Label();
java.awt.Label label15 = new java.awt.Label();
java.awt.Label label17 = new java.awt.Label();
java.awt.Panel panel2 = new java.awt.Panel();
java.awt.Label label18 = new java.awt.Label();
java.awt.Scrollbar controlScrollbar = new java.awt.Scrollbar(Scrollbar.HORIZONTAL,3,1,1,7);
java.awt.Scrollbar failureScrollbar = new java.awt.Scrollbar(Scrollbar.HORIZONTAL,20,1,0,21);
java.awt.Label labelControlL = new java.awt.Label();
java.awt.Label labelControlU = new java.awt.Label();
java.awt.Label labelFailureL = new java.awt.Label();
java.awt.Label labelFailureU = new java.awt.Label();
java.awt.Label label23 = new java.awt.Label();
java.awt.Label label24 = new java.awt.Label();
java.awt.Label label25 = new java.awt.Label();
java.awt.Label label26 = new java.awt.Label();
java.awt.TextField textMTBF = new java.awt.TextField();
java.awt.Label labelMTBF = new java.awt.Label();
java.awt.Panel panel3 = new java.awt.Panel();
java.awt.Button button1 = new java.awt.Button();
java.awt.Label labelReps = new java.awt.Label();
java.awt.TextField textReps = new java.awt.TextField();
//}}

//{{DECLARE_MENU
java.awt.MenuBar mainMenuBar = new java.awt.MenuBar();
java.awt.Menu menu1 = new java.awt.Menu();
java.awt.MenuItem newMenuItem = new java.awt.MenuItem();
java.awt.MenuItem openMenuItem = new java.awt.MenuItem();
java.awt.MenuItem saveMenuItem = new java.awt.MenuItem();
java.awt.MenuItem saveAsMenuItem = new java.awt.MenuItem();
java.awt.MenuItem separatorMenuItem = new java.awt.MenuItem();
java.awt.MenuItem exitMenuItem = new java.awt.MenuItem();
java.awt.Menu menu2 = new java.awt.Menu();
java.awt.MenuItem cutMenuItem = new java.awt.MenuItem();
java.awt.MenuItem copyMenuItem = new java.awt.MenuItem();
java.awt.MenuItem pasteMenuItem = new java.awt.MenuItem();
java.awt.Menu menu3 = new java.awt.Menu();
java.awt.MenuItem aboutMenuItem = new java.awt.MenuItem();
//}}

class SymWindow extends java.awt.event.WindowAdapter
{
    public void windowClosing(java.awt.event.WindowEvent event)
    {
        Object object = event.getSource();
        if (object == JsGui.this)

```

```

        JsfGui_WindowClosing(event);
    }
}

void JsfGui_WindowClosing(java.awt.event.WindowEvent event)
{
    // to do: code goes here.

    JsfGui_WindowClosing_Interaction1(event);
}

void JsfGui_WindowClosing_Interaction1(java.awt.event.WindowEvent event)
{
    try {
        // QuitDialog Create and show as modal
        (new QuitDialog(this, true)).setVisible(true);
    } catch (Exception e) {
    }
}

class SymAction implements java.awt.event.ActionListener
{
    public void actionPerformed(java.awt.event.ActionEvent event)
    {
        Object object = event.getSource();
        if (object == openMenuItem)
            openMenuItem_ActionPerformed(event);
        else if (object == aboutMenuItem)
            aboutMenuItem_ActionPerformed(event);
        else if (object == exitMenuItem)
            exitMenuItem_ActionPerformed(event);
        else if (object == button1)
            button1_ActionPerformed(event);
    }
}

void openMenuItem_ActionPerformed(java.awt.event.ActionEvent event)
{
    // to do: code goes here.

    openMenuItem_ActionPerformed_Interaction1(event);
}

void openMenuItem_ActionPerformed_Interaction1(java.awt.event.ActionEvent event)
{
    try {
        // OpenFileDialog Create and show as modal
        int defMode = openFileDialog1.getMode();
        String defTitle = openFileDialog1.getTitle();
        String defDirectory = openFileDialog1.getDirectory();
        String defFile = openFileDialog1.getFile();

        openFileDialog1 = new java.awt.FileDialog(this, defTitle, defMode);
        openFileDialog1.setDirectory(defDirectory);
        openFileDialog1.setFile(defFile);
        openFileDialog1.setVisible(true);
    } catch (Exception e) {
    }
}

void aboutMenuItem_ActionPerformed(java.awt.event.ActionEvent event)
{
    // to do: code goes here.

```

```

        aboutMenuItem_ActionPerformed_Interaction1(event);
    }

    void aboutMenuItem_ActionPerformed_Interaction1(java.awt.event.ActionEvent event)
    {
        try {
            // AboutDialog Create and show as modal
            (new AboutDialog(this, true)).setVisible(true);
        } catch (Exception e) {
        }
    }

    void exitMenuItem_ActionPerformed(java.awt.event.ActionEvent event)
    {
        // to do: code goes here.

        exitMenuItem_ActionPerformed_Interaction1(event);
    }

    void exitMenuItem_ActionPerformed_Interaction1(java.awt.event.ActionEvent event)
    {
        try {
            // QuitDialog Create and show as modal
            (new QuitDialog(this, true)).setVisible(true);
        } catch (Exception e) {
        }
    }

    class SymAdjustment implements java.awt.event.AdjustmentListener
    {
        public void adjustmentValueChanged(java.awt.event.AdjustmentEvent event)
        {
            Object object = event.getSource();
            if (object == weartimeScrollbar)
                weartimeScrollbar_AdjustmentValueChanged(event);
            else if (object == wearrateScrollbar)
                wearrateScrollbar_AdjustmentValueChanged(event);
            else if (object == flightScrollbar)
                flightScrollbar_AdjustmentValueChanged(event);
            else if (object == controlScrollbar)
                controlScrollbar_AdjustmentValueChanged(event);
            else if (object == failureScrollbar)
                failureScrollbar_AdjustmentValueChanged(event);
        }
    }

    /* The following event listeners adjust the labels for each
    * scrollbar to reflect the value of the scrollbar */
    void weartimeScrollbar_AdjustmentValueChanged(java.awt.event.AdjustmentEvent event)
    {
        label5.setText(Integer.toString(weartimeScrollbar.getValue()));
    }

    void wearrateScrollbar_AdjustmentValueChanged(java.awt.event.AdjustmentEvent event)
    {
        label6.setText(Integer.toString(wearrateScrollbar.getValue()));
    }

    void flightScrollbar_AdjustmentValueChanged(java.awt.event.AdjustmentEvent event)
    {
        label15.setText(Integer.toString(flightScrollbar.getValue()));
    }

```



```

void controlScrollbar_AdjustmentValueChanged(java.awt.event.AdjustmentEvent event)
{
    label23.setText(Integer.toString(controlScrollbar.getValue()));
}

void failureScrollbar_AdjustmentValueChanged(java.awt.event.AdjustmentEvent event)
{
    label24.setText(Integer.toString(failureScrollbar.getValue()));
}

void button1_ActionPerformed(java.awt.event.ActionEvent event)
{
    // This event listener is where the sensor signal is actually built.

    // Variable declarations

    int replications;    //number of replications to be run
    int controllimit;    //out of bounds control limit
    int count1;          //count variable used in the reps while loop
    int arrayLife;        //integer value of double comp_life
    int arrayWearin;      //integer value of double wearintime
    int arrayFailure;     //integer value of double failStart
    int arrayFlight;      //integer value of double flighttime
    int signalLoop;       //increments comp_life to run sim in a loop

    double comp_life;     //expo RV draw for component life (MTBF mean)
    double wearintime;    //wearin portion of component life
    double wearTotal;
    double MTBF;          //mean time between failure (ave life)
    double mean;          //mean of sensor signal
    double wearinrate;    //how often is sensor affected by wearin (%)
    double flightrate;    //how often is sensor affected by flight (%)
    double flighttime;    //how long the component is in steady state
    double failure;       //variability of sensor failure (%)
    double failStart;     //when component starts to show failure
    double batchNumber1;  //number of batches for pre-failure signal
    double batchNumber2;  //number of batches for failure signal
    double batchSize;     //signal batch size

    Random MTBFgen = new Random();    //Life of component
    Random failGen = new Random();    //Failure time generator
    //get the values for the static variables from the GUI interface
    replications = Integer.parseInt(textReps.getText());
    controllimit = controlScrollbar.getValue();
    MTBF = Double.valueOf(textMTBF.getText()).doubleValue();
    mean = Double.valueOf(textMean.getText()).doubleValue();
    wearintime = ((wearintimeScrollbar.getValue())/100.0)*MTBF;
    wearinrate = (wearinrateScrollbar.getValue())/100.0;
    flightrate = (flightScrollbar.getValue())/100.0;
    failure = (failureScrollbar.getValue())/1000.0;
    comp_life = Double.valueOf(textLife.getText()).doubleValue();
    wearTotal = wearinrate + flightrate;
    //initialize variables not associated with GUI interface
    batchSize = 20.0;
    signalLoop = (int) comp_life;
    int intBatch = (int) batchSize;

    /*The simulation starts here and runs until the desired number
    of replications have been run*/
    while (signalLoop < 1501) {
        int totalBatches = 0;
        count1 = 0;
        int lifeNominal [] = new int [((signalLoop/intBatch)) * replications];
        int lifeFailure [] = new int [((signalLoop/intBatch)) * replications];
        double lifeBatch [] = new double [((signalLoop/intBatch)) * replications];
    }
}

```

```

while (count1 < replications) {

    /* The following calculation determines when the failure portion of the *
    * signal will begin (when does the component start to fail). I draw *
    * a NORM(0,1) and multiply it by the variance selected with the *
    * variance slider and then adding a mean of .10. Finally the value *
    * is multiplied by the component life to determine how long the *
    * failure signal array will be. */

    failStart = ((MTBFgen.nextGaussian()*failure)+.1)*signalLoop;

    /* Based on the slider values and the failStart calculation the *
    * duration of the three phases of flight is known. To create an *
    * array that has the length of each phase of flight I need the *
    * phase of flight durations to be integers. This requires explicitly *
    * casting the double variables as integers. */

    arrayLife = (int) signalLoop;
    arrayFailure = (int) (failStart);
    arrayWearin = (int) (wearintime);

    /* Based on the random draw and the "failure" slider setting the length *
    * of the failure portion of the signal could potentially be less than *
    * zero. To solve this problem I add the following loop which completes*
    * the action until the failure array is greater than zero. */

    while (arrayFailure <= 0) {
        failStart = ((MTBFgen.nextGaussian()*failure)+.1)*signalLoop;
        arrayFailure = (int)(failStart);
    }
    /* Based on the random draw to calculate the life of the component *
    * there is a possibility that there could be an array out of bounds *
    * error. If there is a real short life draw the failure portion of *
    * the signal still needs to be calculated. After that any remaining *
    * time is given to the wearin portion of the signal. The flight *
    * portion of the signal is set to zero. */

    arrayFlight = arrayLife - arrayFailure - arrayWearin;
    if ((arrayLife - (arrayFailure + arrayWearin)) < 0 ) {
        arrayWearin = arrayLife - arrayFailure;
        arrayFlight = 0;
    }
    /* The following print statements were used for verification *
    ** purposes only. */

    System.out.print ("The component life is " + arrayLife + " ");
    System.out.println("The failure time is " + (arrayLife - arrayFailure) + ".");
    //System.out.print("The wearin array is " + arrayWearin + " ");
    //System.out.println ("The flight array is " + arrayFlight + ".");

    double Sensor1Signal [] = new double [arrayLife]; //total signal array
    double wearinSignal [] = new double [arrayWearin]; //wearin signal array
    double flightSignal [] = new double [arrayFlight]; //flight signal array
    double failureSignal [] = new double [arrayFailure]; //failure signal array

    /* PrefailSignal array will be used to store the wearin and **
    ** flight portions of the signal. */
    //double prefailSignal [] = new double [arrayWearin + arrayFlight];

    /* Create an instance of the SensorSignal class. */

    SensorSignal Sensor1 = new SensorSignal();

    /* Build three signal portions using methods in SensorSignal class. */

    if (arrayWearin != 0){
        wearinSignal = Sensor1.wearinGenerator(mean, arrayWearin, wearTotal);
    }

```

```

}
if (arrayFlight != 0){
    flightSignal = Sensor1.flightGenerator(mean, arrayFlight, flightrate);
}
failureSignal = Sensor1.failureGenerator(mean, arrayFailure, arrayLife);

/* The following three loops combine the three separate signals into *
* one signal by stacking the arrays into one array. */

/* The next three loops put the wearin and flight portions **
** of the signal together. The final loop is used to build an **
** array that will be used to batch the prefail signal. */

for (int z = 0; z < wearinSignal.length; z++) {
    Sensor1Signal[z] = wearinSignal[z];
}
for (int y = 0; y < flightSignal.length; y++) {
    Sensor1Signal[wearinSignal.length + y] = flightSignal[y];
}
for (int x = 0; x < failureSignal.length; x++) {
    Sensor1Signal[wearinSignal.length + flightSignal.length + x] = failureSignal[x];
}
/*
for (int t = 0; t < prefailSignal.length; t++) {
    prefailSignal[t] = Sensor1Signal[t];
} */

/* The following two segments of code generate the number of **
** batches that the signal will be broken into based on a batch**
** size of 10. The if statement increases the number of **
** batches if necessary. */

batchNumber1 = (arrayWearin + arrayFlight + arrayFailure)/batchSize;
int numBatches1 = (int) batchNumber1;
if ((batchNumber1 - numBatches1) > 0) {
    numBatches1 = numBatches1 + 1;
}
/*
batchNumber2 = arrayFailure/10.0;
int numBatches2 = (int) batchNumber2;
if ((batchNumber2 - numBatches2) > 0) {
    numBatches2 = numBatches2 + 1;
}
*/
/* The following array declarations are for the batched **
** signals. nominalNet and failureNet are arrays with 0 **
** and 1 in them respectively. Zero for nominal **
** component not failing and one when the component is **
** failing. */

//double flightBatch [] = new double [numBatches1];
//double failureBatch [] = new double [numBatches2];
double signalBatch [] = new double [numBatches1];
int nominalNet [] = new int [numBatches1];
int failureNet [] = new int [numBatches1];

// Create batched signal
//flightBatch = Sensor1.batchSignal(numBatches1, batchSize, prefailSignal);
//failureBatch = Sensor1.batchSignal(numBatches2, batchSize, failureSignal);
signalBatch = Sensor1.batchSignal(numBatches1, batchSize,
    Sensor1Signal);

/* The following 4 loops build the total batched signal **
** and append a column of 0's and 1's to it to indicate **
** nominal or failure status. */

/*
for (int m = 0; m < flightBatch.length; m++) {

```

```

    signalBatch[m] = flightBatch[m];
}
*/
/* A batch will be assigned as failing if at least 1 of the
   signals in that batch is failing. */

int healthyState;
int failureState;
double prefailBatch;
prefailBatch = (arrayWearin + arrayFlight)/batchSize;
healthyState = (int) prefailBatch;
failureState = numBatches1 - healthyState;

for (int m = 0; m < healthyState; m++) {
    nominalNet[m] = 1;
    failureNet[m] = 0;
}
/*
for (int n = 0; n < failureBatch.length; n++){
    signalBatch[flightBatch.length + n] = failureBatch[n];
}
*/
for (int n = 0; n < failureState; n++){
    nominalNet[healthyState + n] = 0;
    failureNet[healthyState + n] = 1;
}

/* This loop puts the latest replication in an array **
** that will store all the replications. */

for (int i = 0; i < signalBatch.length; i++){
    lifeBatch[(totalBatches + i)] = signalBatch[i];
    lifeNominal[(totalBatches + i)] = nominalNet[i];
    lifeFailure[(totalBatches + i)] = failureNet[i];
}
totalBatches = totalBatches + signalBatch.length;

// First try statement is the raw signal
try {
    FileOutputStream file = new FileOutputStream("val_full.dat");
    BufferedOutputStream buff = new BufferedOutputStream(file);
    DataOutputStream data = new DataOutputStream(buff);
    for (int i = 0; i < Sensor1Signal.length; i++){
        String s [] = new String [Sensor1Signal.length];
        s[i] = Double.toString(Sensor1Signal[i]) + "\n";
        data.writeChars(s[i]);
    }
    data.close();
} catch (IOException e) {
    System.out.println ("Error -- " + e.toString());
}

try {
    FileOutputStream file = new FileOutputStream("val_wear.dat");
    BufferedOutputStream buff = new BufferedOutputStream(file);
    DataOutputStream data = new DataOutputStream(buff);
    for (int i = 0; i < wearinSignal.length; i++){
        String vhtml [] = new String [wearinSignal.length];
        vhtml[i] = Double.toString(wearinSignal[i]) + "\n";
        data.writeChars(vhtml[i]);
    }
    data.close();
} catch (IOException e) {
    System.out.println ("Error -- " + e.toString());
}

try {
    FileOutputStream file = new FileOutputStream("val_flg.dat");
    BufferedOutputStream buff = new BufferedOutputStream(file);
    DataOutputStream data = new DataOutputStream(buff);
    for (int i = 0; i < flightSignal.length; i++){

```

```

String vhm2 [] = new String [flightSignal.length];
vhm2[i] = Double.toString(flightSignal[i])+ "\n";
data.writeChars(vhm2[i]);
data.close();
} catch (IOException e) {
System.out.println ("Error -- " + e.toString());
}
}
try {
FileOutputStream file = new FileOutputStream("val_fail.dat");
BufferedOutputStream buff = new BufferedOutputStream(file);
DataOutputStream data = new DataOutputStream(buff);
for (int i = 0; i < failureSignal.length; i++){
String vhm3 [] = new String [failureSignal.length];
vhm3[i] = Double.toString(failureSignal[i])+ "\n";
data.writeChars(vhm3[i]);
data.close();
} catch (IOException e) {
System.out.println ("Error -- " + e.toString());
}
}
// Second try statement is the batched signal for a single rep
/*try {
FileOutputStream file = new FileOutputStream(signalLoop + "_" + count1 + "_" + count1 + ".dat");
BufferedOutputStream buff = new BufferedOutputStream(file);
DataOutputStream data = new DataOutputStream(buff);
for (int i = 0; i < signalBatch.length; i++){
String s [] = new String [signalBatch.length];
String t [] = new String [signalBatch.length];
String u [] = new String [signalBatch.length];
s[i] = Double.toString(signalBatch[i])+ "\t";
t[i] = Integer.toString(nominalNet[i])+ "\t";
u[i] = Integer.toString(failureNet[i])+ "\n";
data.writeChars(s[i]);
data.writeChars(t[i]);
data.writeChars(u[i]);
}
data.close();
} catch (IOException e) {
System.out.println ("Error -- " + e.toString());
}
*/
count1++;
}
// Third try statement outputs all reps for a given setting
try {
FileOutputStream file = new FileOutputStream("val_ba20.dat");
BufferedOutputStream buff = new BufferedOutputStream(file);
DataOutputStream data = new DataOutputStream(buff);
for (int i = 0; i < lifeBatch.length; i++){
String mem1 [] = new String [lifeBatch.length];
String mem2 [] = new String [lifeBatch.length];
String mem3 [] = new String [lifeBatch.length];
mem1[i] = Double.toString(lifeBatch[i])+ "\t";
mem2[i] = Integer.toString(lifeNominal[i])+ "\t";
mem3[i] = Integer.toString(lifeFailure[i])+ "\n";
data.writeChars(mem1[i]);
data.writeChars(mem2[i]);
data.writeChars(mem3[i]);
}
data.close();
} catch (IOException e) {
System.out.println ("Error -- " + e.toString());
}
}
signalLoop = signalLoop + 1;
if (signalLoop == 301){
signalLoop = 700;
}
if (signalLoop == 701) {
signalLoop = 1100;
}
}

```

```

        if (signalLoop == 1101) {
            signalLoop = 1500;
        }
        if (signalLoop == 1501){
            signalLoop = 2100;
        }
        if (signalLoop == 2101){
            signalLoop = 2700;
        }
        if (signalLoop == 2701) {
            signalLoop = 3700;
        }
        if (signalLoop == 3701) {
            signalLoop = 4900;
        }
        if (signalLoop == 4901){
            signalLoop = 7200;
        }
        if (signalLoop == 7201){
            signalLoop = 14100;
        }
    }

}

}

SensorSignal Class
/* Capt Mike Malley          **
** GOR01M                    **
**                          **
** Last modified: 27 Dec 00   */

import java.io.*;
import java.util.*;

public class SensorSignal {
    // SensorSignal Constructor - creates instance of a sensor signal
    public SensorSignal () {
    }
    /* Method wearinGenerator. This method requires the signal mean, **
    ** a wearinTime, and wearinSens. The variable wearinTime is **
    ** simply the amount of time that the wearinGenerator will be **
    ** used to create a signal. wearinTime becomes the length of the **
    ** array passed back to the main program. wearinSens is the **
    ** sensitivity of the sensor to wearin conditions and is a **
    ** percent. The output of the this method is an array that **
    ** contains the wearin portion of the sensor signal. */

    public double [] wearinGenerator (double mean, int wearinTime,
                                     double wearinSens) {

        /* Variable Declarations. seed1 and seed2 are created using **
        ** Java's built-in Math.random() method so that each **
        ** replication has unique values. gen1 and gen2 are simply **
        ** instances of Java's 48 bit linear congruential RN that **

```

```

** will be used to induce randomness into the sensor signal. **
** storageVar1 simply stores the uniform RN draw that is **
** compared to wearinSens to determine if a wearin "event" **
** occurs. mean_shift is not a shift but replaces the mean **
** when a wearin "event" does occur. Finally, signalCorr is **
** the same as defined in JSFGui. */

long seed1 = (long) (Math.random() * 800000000000000L );
long seed2 = (long) (Math.random() * 6500000000 );
Random gen1 = new Random(seed1);
Random gen2 = new Random(seed2);
double wearSignal[] = new double[wearinTime];
double storageVar1 = 0;
double mean_shift;
double signalCorr = .8;
wearSignal[0] = 500.0 + gen2.nextGaussian();

/* The initial wearSignal uses 500 + NORMAL(0,1) because **
** when the mean is set at 100 the signal steady state is **
** 500. If the mean is changed then, the 500 should also be **
** changed. */

for (int a = 1; a < wearSignal.length; a=a+5) {
    // a=a+5 because the signal is generated in batches of 5
    storageVar1 = gen1.nextDouble();
    // if system experiences a wearin "event"
    if (storageVar1 <= wearinSens){
        mean_shift = mean + (gen2.nextDouble() * 2.25);
        // if next five signals do NOT bust array length
        if ((a+5) < wearSignal.length){
            for (int g = a; g < a+4; g++) {
                wearSignal[g] = mean_shift + signalCorr*wearSignal[g-1] + gen2.nextGaussian();
            }
            wearSignal[a+4] = mean + signalCorr*wearSignal[a-1] + gen2.nextGaussian();
        }
        // (else) next five signals bust array length
        if ((a+5)>= wearSignal.length){
            for (int d = a; d < wearSignal.length; d++){
                wearSignal[d] = mean_shift + signalCorr*wearSignal[d-1] + gen2.nextGaussian();
            }
        }
    }
    // (else) if system operates nominally
    if (storageVar1 > wearinSens) {
        // if next five signals do NOT bust array length
        if ((a+5) < wearSignal.length){
            for (int i = a; i < a+5; i++) {
                wearSignal[i] = mean + signalCorr*wearSignal[i-1] + gen2.nextGaussian();
            }
        }
        // (else) next five signals bust array length
        if ((a+5)>= wearSignal.length){

```

```

        for (int d = a; d < wearSignal.length; d++){
            wearSignal[d] = mean + signalCorr*wearSignal[d-1] + gen2.nextGaussian();
        }
    }
}

return wearSignal;
}

```

```

/* Method flightGenerator. This method requires the signal mean, **
** a flightTime, and flightSens. The variable flightTime is **
** simply the amount of time that the flightGenerator will be **
** used to create a signal. flightTime becomes the length of the **
** array passed back to the main program. flightSens is the **
** sensitivity of the sensor to flight conditions and is a **
** percent. The output of the this method is an array that **
** contains the flight portion of the sensor signal. */

```

```

public double [] flightGenerator (double mean, int flightTime,
                                double flightSens){

```

```

    /* Variable Declarations. seed1 and seed2 are created using **
    ** Java's built-in Math.random() method so that each **
    ** replication has unique values. gen3 and gen4 are simply **
    ** instances of Java's 48 bit linear congruential RN that **
    ** will be used to induce randomness into the sensor signal. **
    ** storageVar2 simply stores the uniform RN draw that is **
    ** compared to wearinSens to determine if a wearin "event" **
    ** occurs. mean_shift is not a shift but replaces the mean **
    ** when a wearin "event" does occur. Finally, signalCorr is **
    ** the same as defined in JSFGui. */

```

```

    long seed1 = (long) (Math.random() * 78000 );
    long seed2 = (long) (Math.random() * 4589000000000000L );
    Random gen3 = new Random(seed1);
    Random gen4 = new Random(seed2);
    double flightSignal[] = new double[flightTime];
    double storageVar2 = 0;
    double mean_shift;
    double signalCorr = .8;
    flightSignal[0] = 500.0 + gen4.nextGaussian();

```

```

    /* The initial flightSignal uses 500 + NORMAL(0,1) because **
    ** when the mean is set at 100 the signal steady state is **
    ** 500. If the mean is changed then, the 500 should also be **
    ** changed. */

```

```

    for (int b = 1; b < flightSignal.length; b=b+5) {
        // b=b+5 because the signal is generated in batches of 5
        storageVar2 = gen3.nextDouble();
        // if system experiences a flight "event"
        if (storageVar2 <= flightSens){

```



```

mean_shift = mean + (gen3.nextDouble() * 2.25);
// if next five signals do NOT bust array length
if ((b+5) < flightSignal.length){
    for (int e = b; e < b+4; e++) {
        flightSignal[e] = mean_shift + signalCorr * flightSignal[e-1] + gen4.nextGaussian();
    }
    flightSignal[b+4] = mean + signalCorr*flightSignal[b-1] + gen4.nextGaussian();
}
// (else) next five signals bust array length
if ((b+5) >= flightSignal.length){
    for (int c = b; c < flightSignal.length; c++){
        flightSignal[c] = mean_shift + signalCorr*flightSignal[c-1] + gen4.nextGaussian();
    }
}
}
// (else) if system operates nominally
if (storageVar2 > flightSens) {
    // if next five signals do NOT bust array length
    if ((b+5) < flightSignal.length){
        for (int f = b; f < b+5; f++) {
            flightSignal[f] = mean + signalCorr*flightSignal[f-1] + gen4.nextGaussian();
        }
    }
    // (else) next five signals bust array length
    if ((b+5) >= flightSignal.length){
        for (int h = b; h < flightSignal.length; h++){
            flightSignal[h] = mean + signalCorr*flightSignal[h-1] + gen4.nextGaussian();
        }
    }
}
}
}

return flightSignal;
}

```

```

/* Method failureGenerator. This method requires the signal mean,**
** a failureTime, and compLife. The variable failureTime is **
** simply the amount of time that the failureGenerator will be **
** used to create a signal. failureTime becomes the length of **
** the array passed back to the main program. compLife is the **
** life of the component which is used to increment the signal **
** towards failure. The output of the this method is an array **
** that contains the flight portion of the sensor signal. */

```

```

public double [] failureGenerator (double mean, int failureTime,
int compLife) {

```

```

/* Variable Declarations. seed is created using Java's **
** built-in Math.random() method so that each replication **
** has unique values. gen5 is simply an instance of Java's **
** 48 bit linear congruential RN that will be used to **

```

```

    /** induce randomness into the sensor signal. Limit is
    /** arbitrarily set to 510 because the steady-state signal is
    /** 500. The point is I had to choose something high enough
    /** for the Neural Network to detect failure. Increment is
    /** the stepsize that will get from the baseline 500 to 510 in
    /** failureTime.
    */

    long seed = (long) (Math.random() * 687420000000L );
    Random gen5 = new Random(seed);
    double limit = 510.0;
    double increment = (limit-500.0)/failureTime;
    double failureSignal[] = new double[failureTime];
    failureSignal[0] = 500.0 + increment;
    for (int c = 1; c < failureSignal.length; c++) {
        failureSignal[c] = failureSignal[c-1] + (increment * (0.7 + (0.6 * gen5.nextDouble())));
    }
    return failureSignal;
}

/* Method batchSignal. This method batches the raw signal into
** numBatches with each batch having batchSize elements. The
** method returns an array whose elements are the batched signal. */

public double [] batchSignal (int numBatches, double batchSize, double signalArray []) {

    /* Variable declarations. count2, count3, and count4 are
    /** simply count variables. lastBatch is the last batch to be
    /** calculated. batchSignal will be returned to the main
    /** program.
    */

    int count2 = 0;
    int lastBatch = numBatches - 1;
    double batchSignal[] = new double [numBatches];
    // Loop to batch until numBatches has been achieved
    while (count2 < numBatches) {
        double sum = 0;
        double average = 0;
        int count3 = (int) (count2 * batchSize);
        /* If this is the last batch then the sum and average
        /** calculations need to be calculated differently.
        */
        if (count2 == lastBatch) {
            int count4 = 0;
            while (count3 < signalArray.length) {
                sum = sum + signalArray[count3];
                count4++;
                count3++;
            }
            average = sum/count4;
        }
        // (Else) when this is not the last batch
        if (count2 != lastBatch) {
            while (count3 < ((count2*batchSize) + batchSize)) {
                sum = sum + signalArray[count3];
                count3++;
            }
        }
    }
}

```

```
        }  
        average = sum/batchSize;  
    }  
    batchSignal[count2] = average;  
    count2++;  
}  
return batchSignal;  
}
```

```
}
```

Appendix B. ALSim PHM Code

//Source file: C:/ljsf/PHM.java

```
import com.threadtec.silk.random.*;
import com.threadtec.silk.statistics.*;
```

```
/**
```

```
 * This class represents the PHM system installed in each of the JSF aircraft
```

```
 **/
```

```
public class PHM
```

```
{
```

```
 // double detectionFactor = 0.01; //0.1; // represents the 5% PHM detection factor
```

```
 double detectionFactor = 0.05; //0.1; // represents the 5% PHM detection factor
```

```
 // double detectionFactor = 0.1; //0.1; // represents the 5% PHM detection factor
```

```
 double nominalMean = 0.944;
```

```
 double nominalStd = 0.0141;
```

```
 double[] probabilityFA = { .0156, .0444, .0636,
                             .0778, .1089, .1156,
                             .1422, .1822, .2714,
                             .3778};
```

```
 double hoursLRU1 [] = {0, 100.0, 220.0, 340.0,
                        490.0, 675.0, 875.0,
                        1150.0, 1550.0, 2250.0,
                        4575.0};
```

```
 double hoursLRU2 [] = {0, 85.0, 190.0, 300.0,
                        430.0, 590.0, 775.0,
                        1000.0, 1350.0, 2000.0,
                        4000.0};
```

```
 double hoursLRU3 [] = {0, 50.0, 100.0, 160.0,
                        225.0, 300.0, 400.0,
                        525.0, 700.0, 1000.0,
                        2000.0};
```

```
 double whenFailed = 0.0; // when during flight failure is detected
```

```
 double whenDetected = 0.0; // when prognostic kicks-in
```

```
// double detectionFactor = 0.050; //0.05; // represents the 5% PHM detection factor
 // PHM will detect failure within 5% of actual failure
```

```
 private boolean notProcessedLRU1 = true; // true --> not yet processed by PHM
```

```
 private boolean notProcessedLRU2 = true;
```

```
 private boolean notProcessedLRU3 = true;
```

```
 /** ----- **/
```

```
 * METHODS (OPERATIONS)
```

```
 *** ----- **/
```

```

    /**
    * null constructor
    */
    public PHM() {}

    /* time during flight a failure occurred -- delta after take-off
    * KEY: A part failed, when did failure occur?
    */
    public void doPHM(double flightLength, int id)
    {
        setFailureTime(flightLength, id); // determine when failure occurred
    }
    public void doPHM(double flightLength, int id, LRU1 LRU)
    {
        if (notProcessedLRU1)
        {
            setPrognostic(LRU,id); // future equipment failure
            setDetectionTime(LRU,id); // determine when the prognostic is done
        }
    }
    public void doPHM(double flightLength, int id, LRU2 LRU)
    {
        if (notProcessedLRU2)
        {
            setPrognostic(LRU,id); // future equipment failure
            setDetectionTime(LRU,id); // determine when the prognostic is done
        }
    }
    public void doPHM(double flightLength, int id, LRU3 LRU)
    {
        if (notProcessedLRU3)
        {
            setPrognostic(LRU,id); // future equipment failure
            setDetectionTime(LRU,id); // determine when the prognostic is done
        }
    }

    /**
    * this method will only be done if failure occurred
    * reason: can't fix a/c before or DURING FLIGHT
    */
    private void setFailureTime(double flightLength, int id)
    {
        // ASSUME: PHM detection of failed parts are Uniformly distributed (during flight)
        Uniform uniPHMdetect = new Uniform( 0.0, flightLength );
        whenFailed = uniPHMdetect.sample(); //draw a sample
        JDISclass.trackTimeFailed[id]= whenFailed;
    }

    /**
    * allows maintenance to reset PHM
    */
    public void clearFailureTime(int id)
    {
        JDISclass.trackTimeFailed[id]= 0.0;
    }

```

```

/**
 * the following methods are PHM's guess on when the LRU fails
 */
private void setPrognostic(LRU1 LRU, int id)
{
    // ASSUME: PHM is 100% accurate, it can predict the exact
    // time the equipment fails
    whenFailed = LRU.getFailureTime();
    JDISClass.trackTimePrognostic[id][0]= whenFailed;
    notProcessedLRU1 = false;
}
private void setPrognostic(LRU2 LRU, int id)
{
    // ASSUME: PHM is 100% accurate, it can predict the exact
    // time the equipment fails
    whenFailed = LRU.getFailureTime();
    JDISClass.trackTimePrognostic[id][1]= whenFailed;
    notProcessedLRU2 = false;
}
private void setPrognostic(LRU3 LRU, int id)
{
    // ASSUME: PHM is 100% accurate, it can predict the exact
    // time the equipment fails
    // System.out.println ( "\n at PHM.setPrognostic LRU3 not processed?: ");
    whenFailed = LRU.getFailureTime();
    JDISClass.trackTimePrognostic[id][2]= whenFailed;
    notProcessedLRU3 = false;
}
/**
 * these methods simulate the time of prognostic
 * reason: this is necessary to determine WHEN replacement parts will be ordered
 */
private void setDetectionTime(LRU1 LRU, int id)
{
    /*
    Uniform LRU1FA = new Uniform (0.0, 1.0);
    Uniform timeFA = new Uniform (0.0, .9 * LRU.getFailureTime());
    double drawFA = LRU1FA.sample();

    // ASSUME: PHM detection of failed parts are assumed to be less
    // than x% of the actual failure time
    Normal drawLRU1 = new Normal (nominalMean, nominalStd, 10000);
    whenDetected = drawLRU1.sample() * LRU.getFailureTime();

    if (whenDetected > 4575)
        whenDetected = 4575;

    int arrayCount = 0;

    if (whenDetected <= 100 && whenDetected >= 0)
        arrayCount = 0;
    if (whenDetected <= 220 && whenDetected > 100)
        arrayCount = 1;
    if (whenDetected <= 340 && whenDetected > 220)

```

```

    arrayCount = 2;
    if (whenDetected <= 490 && whenDetected > 340)
        arrayCount = 3;
    if (whenDetected <= 675 && whenDetected > 490)
        arrayCount = 4;
    if (whenDetected <= 875 && whenDetected > 675)
        arrayCount = 5;
    if (whenDetected <= 1150 && whenDetected > 875)
        arrayCount = 6;
    if (whenDetected <= 1550 && whenDetected > 1150)
        arrayCount = 7;
    if (whenDetected <= 2250 && whenDetected > 1550)
        arrayCount = 8;
    if (whenDetected <= 4575 && whenDetected > 2250)
        arrayCount = 9;

    if (drawFA <= probabilityFA[arrayCount]) {
        whenDetected = timeFA.sample();
        if (whenDetected > 0.0) JDISclass.trackTimeDetected[id][0]= whenDetected; // update JDIS
        else JDISclass.trackTimeDetected[id][0]= Simulation.time;
    }*/
    // This is Rene's original code
    double PHMfunction = detectionFactor*(LRU.getFailureTime());
    whenDetected = LRU.getFailureTime() - PHMfunction;
    //
    if (whenDetected > 0.0) JDISclass.trackTimeDetected[id][0]= whenDetected; // update JDIS
    else JDISclass.trackTimeDetected[id][0]= Simulation.time;
}
private void setDetectionTime(LRU2 LRU, int id)
{
    /*
    Uniform LRU2FA = new Uniform (0.0, 1.0);
    Uniform timeFA = new Uniform (0.0, .9 * LRU.getFailureTime());
    double drawFA = LRU2FA.sample();

    // ASSUME: PHM detection of failed parts are assumed to be less
    // than x% of the actual failure time

    Normal drawLRU2 = new Normal (nominalMean, nominalStd, 1000000);
    whenDetected = drawLRU2.sample() * LRU.getFailureTime();

    if (whenDetected > 4000)
        whenDetected = 4000;

    int arrayCount = 0;

    if (whenDetected <= 85 && whenDetected >= 0)
        arrayCount = 0;
    if (whenDetected <= 190 && whenDetected > 85)
        arrayCount = 1;
    if (whenDetected <= 300 && whenDetected > 190)
        arrayCount = 2;
    if (whenDetected <= 430 && whenDetected > 300)

```

```

        arrayCount = 3;
    if (whenDetected <= 590 && whenDetected > 430)
        arrayCount = 4;
    if (whenDetected <= 775 && whenDetected > 590)
        arrayCount = 5;
    if (whenDetected <= 1000 && whenDetected > 775)
        arrayCount = 6;
    if (whenDetected <= 1350 && whenDetected > 1000)
        arrayCount = 7;
    if (whenDetected <= 2000 && whenDetected > 1350)
        arrayCount = 8;
    if (whenDetected <= 4000 && whenDetected > 2000)
        arrayCount = 9;

    if (drawFA <= probabilityFA[arrayCount]) {
        whenDetected = timeFA.sample();
        if (whenDetected > 0.0) JDISclass.trackTimeDetected[id][0]= whenDetected; // update JDIS
        else JDISclass.trackTimeDetected[id][0]= Simulation.time;
    }*/
    //
    double PHMfunction = detectionFactor*(LRU.getFailureTime());
    whenDetected = LRU.getFailureTime() - PHMfunction;
    //
    if (whenDetected > 0.0) JDISclass.trackTimeDetected[id][0]= whenDetected; // update JDIS
    else JDISclass.trackTimeDetected[id][0]= Simulation.time;
}
private void setDetectionTime(LRU3 LRU, int id)
{
    /*
    Uniform LRU3FA = new Uniform (0.0, 1.0);
    Uniform timeFA = new Uniform (0.0, .9 * LRU.getFailureTime());
    double drawFA = LRU3FA.sample();

    // ASSUME: PHM detection of failed parts are assumed to be less
    // than x% of the actual failure time

    Normal drawLRU3 = new Normal (nominalMean, nominalStd, 100);
    whenDetected = drawLRU3.sample() * LRU.getFailureTime();

    if (whenDetected > 2000)
        whenDetected = 2000;

    int arrayCount = 0;

    if (whenDetected <= 50 && whenDetected >= 0)
        arrayCount = 0;
    if (whenDetected <= 100 && whenDetected > 50)
        arrayCount = 1;
    if (whenDetected <= 160 && whenDetected > 100)
        arrayCount = 2;
    if (whenDetected <= 225 && whenDetected > 160)
        arrayCount = 3;
    if (whenDetected <= 300 && whenDetected > 225)

```



```

        arrayCount = 4;
    if (whenDetected <= 400 && whenDetected > 300)
        arrayCount = 5;
    if (whenDetected <= 525 && whenDetected > 400)
        arrayCount = 6;
    if (whenDetected <= 700 && whenDetected > 525)
        arrayCount = 7;
    if (whenDetected <= 1000 && whenDetected > 700)
        arrayCount = 8;
    if (whenDetected <= 2000 && whenDetected > 1000)
        arrayCount = 9;

    if (drawFA <= probabilityFA[arrayCount]) {
        whenDetected = timeFA.sample();
        if (whenDetected > 0.0) JDISclass.trackTimeDetected[id][0]= whenDetected; // update JDIS
            else JDISclass.trackTimeDetected[id][0]= Simulation.time;
    }*/
    //
    double PHMfunction = detectionFactor*(LRU.getFailureTime());
    whenDetected = LRU.getFailureTime() - PHMfunction;
    //
    if (whenDetected > 0.0) JDISclass.trackTimeDetected[id][2]= whenDetected; // update JDIS
        else JDISclass.trackTimeDetected[id][2]= Simulation.time;
}
/**
 * allows maintenance to reset PHM's prognostics after
 * replacement parts are installed
 */
public void resetPrognostic(LRU1 LRU)
{
    notProcessedLRU1 = true;
}
public void resetPrognostic(LRU2 LRU)
{
    notProcessedLRU2 = true;
}
public void resetPrognostic(LRU3 LRU)
{
    notProcessedLRU3 = true;
}
}

```

Bibliography

- Atlas, Les, George Bloor, Tom Brotherton, Larry Howard, Link Jaw, Greg Kacprzyński, Gabor Karsai, Ryan Mackey, Jay Mesick, Rick Reuter, and Mike Roemer. "An Evolvable Tri-Reasoner IVHM System." Boeing Company publication. 1-15. Copyright 1999, The Boeing Company.
- Bauer, Ken. Class handout, Artificial Neural Network. School of Engineering and Management, Air Force Institute of Technology, Wright-Patterson AFB OH, September 2000.
- "Condor Launches New Data Systems & Solutions Internet Aeroengine Service." Data Systems and Solutions new release. http://www.ds-s.com/News/News_07. Released 26 July 2000.
- "In-Flight Health Checks aid Aeroengines Maintenance." Data Systems and Solutions news release. http://www.ds-s.com/News/News_04. Released 29 July 1999.
- Blemel, Kenneth. "Dynamic Autonomous Test Systems for Prognostic Health Management." (AD-A355365). 1998.
- Felke, Tim. "Application of Model-Based Diagnostic Technology on the Boeing 777 Airplane," AIAA/IEEE Digital Avionics Systems Conference. 633-639. New York: IEEE Press, 1994.
- Hough, Michael. "The Affordable Solution - JSF," PowerPoint presentation. 1-25. <http://www.jast.mil>. 1999.
- JetSCAN Oil Analysis System. http://www.ds-s.com/Products/PS_JetSCAN. Data System and Solutions, 27 August 2000.
- JSF Program Office. "Joint Strike Fighter Autonomic Logistics Briefing," PowerPoint presentation. 1-24. <http://www.jast.mil/assets/multimedia/autologistics.pdf>. 15 February 2000.
- Keller, Terry and Mark Eslinger. Digital Signal Processing Technology Transfer for Machine Monitoring. United States Air Force contract F33615-98-C-2873.

Frontier Technology, Incorporated, 4141 Colonel Glenn Highway #140,
Beavercreek OH, 30 August 1999.

Knapp, G. M. and H. P. Wang. "Automated tactical maintenance planning based on machine monitoring," International Journal of Production Research, 34(3): 753-765 (March 1996)

Pomfret, Chris. "Search for an Oil or Vibration Related Solution to the GE#4 Bearing Problem". Facsimile communication between Aerospace Business Development Associates Incorporated and Air Force Research Laboratory Propulsion Directorate. 1838, 6 January 2000.

Powrie, H. E. G. and C. E. Fisher. "Engine Health Monitoring: Towards Total Prognostics," IEEE Aerospace Applications Conference Proceedings. 11-20. Los Alamitos CA: IEEE Press, 1999.

Rebulanan, Rene. Simulation of the Joint Strike Fighter's Autonomic Logistics System Using the Java Programming Language. MS thesis, AFIT/GOR/ENS/00M-19. School of Engineering and Management, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 2000.

Scheuren, W. "Safety & The Military Aircraft Joint Strike Fighter Prognostics & Health Management." 34th AIAA/ASME/SAE/ASEE Joint Propulsion Conference and Exhibit July 13-15, 1998, Cleveland, Ohio: 1-7. American Institute of Aeronautics and Astronautics, 1801 Alexander Bell Drive, Suite 500, Reston VA, July 1998.

Szczerbicki, Edward and Warren White. "System modeling and simulation for predictive maintenance," Cybernetics and Systems, 29: 481-498 (1998).

US Air Force Research Laboratory Propulsion Directorate. Contract F33615-98-C-2873 with Frontier Technology Incorporated. Wright-Patterson AFB OH, 30 Aug 99.

Walls, Michael, Mark Thomas, and Thomas Brady. "Improving system maintenance decisions: a value of information framework," The Engineering Economist, 44: 151-166 (February 1999).

Vita

Captain Michael E. Malley was born in Bloomington, IL. He attended Port St Lucie High School in Port St Lucie, FL and graduated at the top of his class in 1992. After graduation he entered the United States Air Force Academy and graduated from Cadet Squadron 23 in 1996 with a Bachelor Science Degree in Mechanical Engineering. He married shortly after graduation. His first duty assignment was with the Airborne Laser System Program Office at Kirtland AFB, NM as a program analyst. After one year in the Program Office he moved to a system engineering position.

After graduation from AFIT Captain Malley will be assigned to the National Reconnaissance Office in Chantilly, VA.

REPORT DOCUMENTATION PAGE				<i>Form Approved OMB No. 074-0188</i>	
The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of the collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.					
PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.					
1. REPORT DATE (DD-MM-YYYY) 20-03-2001		2. REPORT TYPE Master's Thesis		3. DATES COVERED (From - To) Jun 2000 - Mar 2001	
4. TITLE AND SUBTITLE A METHODOLOGY FOR SIMULATING THE JOINT STRIKE FIGHTER'S PROGNOSTICS AND HEALTH MANAGEMENT SYSTEM				5a. CONTRACT NUMBER	
				5b. GRANT NUMBER	
				5c. PROGRAM ELEMENT NUMBER	
6. AUTHOR(S) Malley, Michael E, Capt, USAF				5d. PROJECT NUMBER 99-410	
				5e. TASK NUMBER	
				5f. WORK UNIT NUMBER	
7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/ENS) 2950 P Street, Building 640 WPAFB OH 45433-7765				8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GOR/ENS/01M-11	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Dayton Area Graduate Studies Institute Attn: Dr Frank Moore 3155 Research Blvd, Suite 205 Kettering, OH 45420 (937) 257-1346				10. SPONSOR/MONITOR'S ACRONYM(S)	
				11. SPONSOR/MONITOR'S REPORT NUMBER(S)	
12. DISTRIBUTION/AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.					
13. SUPPLEMENTARY NOTES					
14. ABSTRACT <p>The Autonomic Logistics System Simulation (ALSim) model was developed to provide decision makers a tool to make informed decisions regarding the Joint Strike Fighter's (JSF) Autonomic Logistics System (ALS). The ALS provides real-time maintenance information to ground maintenance crews, supply depots, and air planners to efficiently manage the availability of JSF aircraft. This thesis effort focuses on developing a methodology to model the Prognostics and Health Management (PHM) component of ALS. The PHM component of JSF monitors the aircraft status.</p> <p>To develop a PHM methodology to use in ALSim a neural network approach is used. Notional JSF prognostic signals were generated using an interactive Java application, which were then used to build and train a neural network. The neural network is trained to predict when a component is healthy and/or failing. The results of the neural network analysis are meaningful failure detection times and false alarm rates. The analysis presents a batching approach to train the neural network, and looks at the sensitivity of the results to batch size and the neural network classification rule used. The final element of the research is implementing the PHM methodology in the (ALSim).</p>					
15. SUBJECT TERMS Aircraft maintenance; simulation; neural nets; Joint Strike Fighter; prognostic modeling					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT UU	18. NUMBER OF PAGES 117	19a. NAME OF RESPONSIBLE PERSON Miller, J.O., Lt Col, USAF AFIT/ENS
a. REPORT U	b. ABSTRACT U	c. THIS PAGE U			19b. TELEPHONE NUMBER (Include area code) (937) 255-6565, ext 4326 John.Miller@afit.af.edu

Standard Form 298 (Rev. 8-98)
 Prescribed by ANSI Std. Z39-18

	<i>Form Approved OMB No. 074-0188</i>
--	---